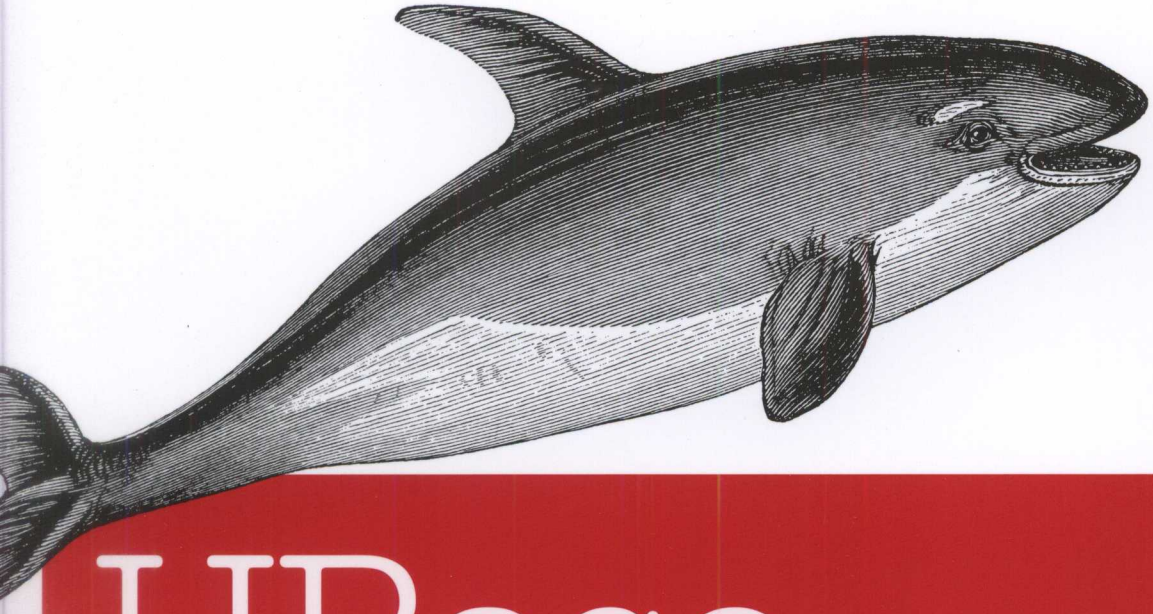
O'REILLY®



HBase 应用架构

Architecting HBase Applications

Jean-Marc Spaggiari, Kevin O'Dell 著
陈敏敏 夏锐 陈其生 译

中国电力出版社

HBase应用架构

Jean-Marc Spaggiari Kevin O'Dell 著

陈敏敏 夏锐 陈其生 译

Beijing • Boston • Farnham • Sebastopol • Tokyo O'REILLY®

O'Reilly Media, Inc. 授权中国电力出版社出版

中国电力出版社

图书在版编目（CIP）数据

HBase应用架构 / (美) 吉恩－马克·斯帕加里（Jean-Marc Spaggiari），（美）凯文·奥戴尔（Kevin O'Dell）著；陈敏敏，夏锐，陈其生译. — 北京：中国电力出版社，2017.9

书名原文：Architecting HBase Applications

ISBN 978-7-5198-1121-1

I. ①H… II. ①吉… ②凯… ③陈… ④夏… ⑤陈… III. ①计算机网络－信息存贮 IV. ①TP393

中国版本图书馆CIP数据核字(2017)第219369号

北京市版权局著作权合同登记 图字：01-2017-4340号

出版发行：中国电力出版社

地　址：北京市东城区北京站西街19号（邮政编码100005）

网　址：http://www.cepp.sgcc.com.cn

责任编辑：刘 炽（liuchi1030@163.com）

责任校对：常燕昆

装帧设计：Karen Montgomery，张 健

责任印制：蔺义舟

印　刷：北京天宇星印刷厂

版　次：2017年9月第一版

印　次：2017年9月北京第一次印刷

开　本：750毫米 × 980 毫米　16开本

印　张：14.5　980

字　数：273千字

印　数：0001—3000册

定　价：48.00元

O'Reilly Media, Inc.介绍

O'Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O'Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O'Reilly的发展充满了对创新的倡导、创造和发扬光大。

O'Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O'Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O'Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O'Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O'Reilly Radar博客有口皆碑。”

——Wired

“O'Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O'Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O'Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

译者序

HBase是Hadoop Database的简称，基于Google Bigtable实现的开源、分布式、可伸缩的列式存储数据库，自其诞生之日起，就受到业界的追捧，而今已成为了Hadoop生态圈和各公司大数据平台不可或缺的组成部分。时间序列数据库Opentsdb、阿里的HQueue、OLAP分析引擎Kylin等很多大数据开源项目也都是基于HBase的。

本书是HBase应用架构的手册指南，它涵盖了HBase实践应用的各个方面，从HBase底层的基本原理到整个生态系统，从最初的应用实例到性能优化，以及后续的疑难问题的定位与排错等方面。同时，作者也列举了大量的实例场景及实现代码，读者都可直接从中获取实际应用原型。

本书主要由陈敏敏（苏宁云商大数据中心总监）、夏锐（奇虎360技术专家）、陈其生（唯品会技术专家）翻译，由于译者水平所限，翻译难免有疏漏之处，恳广大读者给予批评指正。

译者

感谢我父亲，我希望你能看到这本书……

——Jean-Marc Spaggiari

感谢我日思夜想的母亲，感谢每天一直陪在我身边的父亲，以及我美丽的妻子Melanie和女儿Scotland，是她们的陪伴帮助我度过枯燥且漫长的写作生活。

——Kevin O'Dell

目录

第三部分 疑难问题的定位和排错

序

在我25年软件从业生涯里，经历了许多颠覆性的变化：互联网、万维网、大型机、C/S架构模型等。我曾经在一个保障炼油厂安全的软件研发团队工作。我们40人的团队共享唯一的一台DEC VAX机器，而且性能也远不及我现在使用的手机。

我还记得20世纪90年代初的一天，我们被安排接收新的机器。要更换的机器位于三楼，是一个很大很重的像洗衣机大小的庞然大物。我们一群人在“机房”里等着，想看看他们是怎么把新机器搬上楼梯的。我们可以想象一下一个巨大的起重机，外面的街道被封锁的场景……总之，这是一个浩大的工程！

但实际上出乎意料。一个男的提着一个小箱子走进房间。他把它放在那台像老式“洗衣机”的机器的顶部，换了几根电缆，做了一些测试，便离开了。这样，就完成了！

这是技术产业带来的喜悦：如果我们愿意学习新事物，并与之同行，我们将永远不会感到厌倦，不断地遇见惊喜。几年前似乎不可能的事情突然会变得司空见惯。

大数据就是这样在变化，大数据无处不在。由谷歌公司的Google File System和BigTable引发的技术变革几乎影响了所有的高科技公司、银行和政府。无论是好是坏，这些系统直接或间接地触及了地球上几乎每个人的生活。

和它之前的BigTable一样，Apache HBase已经在大数据生态系统中处于独特的地位：它在一个不变的分布式文件系统中支持了一种可更新的、无限扩展的数据集存储。因此，它构建了纯文件存储和OLTP / OLAP数据库之间的桥梁。

HBase无处不在：Facebook、苹果、Salesforce.com、Adobe、雅虎、Bloomberg、华为、Gap和许多其他公司都在使用它。谷歌采用了HBase API为其提供公共云Bigtable，这也是HBase受欢迎的一个证据。

尽管HBase无处不在，但它也不是即插即用。分布式系统很难，诸如分区容忍性、一致性和可用性等在每个讨论区都变成了大家热议的话题，不久之后更深奥的术语如热点(hotspotting)和数据加盐(salting)存储也出现了。成百上千台机器的水平扩展都需要经历痛苦的数据平衡过程，这些平衡使得我们更加难以有效地使用这些系统，HBase也不例外。

在混迹于HBase和Hadoop社区的这段岁月里，我已获得了这些挑战的第一手经验。必须仔细设计和构建这些用例，以发挥HBase的优势。

这本书由两位内部人士撰写，他们一直在现场做支持。本书是一个非常有帮助的指南，它详细介绍了如何基于HBase架构来设计应用程序，并能够支持数千台机器的集群扩展。

如果你正在构建或计划构建高扩展性及高可靠性的新应用程序，那么这本书就是为你设计的。Jean-Marc 和Kevin已经看到这一切：使用案例，人们犯的错误以及单服务器系统的设计臆想。最重要的是，他们知道哪些是运行正常的，如何解决那些运行不正常的情况，以及如何理解这里面的本质。

—— Lars Hofhansl，
HBase的提交者和成员
项目管理委员会
Apache基金会成员Salesforce.com副总裁

前言

随着Hadoop越来越受到很多人的欢迎，其生态系统也充满活力，包括广泛使用的工具，如Hive、Spark、Impala及 HBase。这本书着重于工具Apache HBase，它构建于Hadoop分布式文件系统（HDFS）之上，具有可扩展性、容错性、低延迟等特征。HBase整合了Hadoop的水平扩展能力和实时数据服务两方面的优势。在规模方面，HBase允许从单个集群中进行每秒钟数百万次的读写操作，同时仍可保持Hadoop所有应用的可用。HBase迅速普及，现在已经为世界上一些较大的Hadoop集群部署提供了支持，如Apple、Salesforce.com及 Facebook。

然而，开始使用HBase是一个艰巨的任务。虽然有许多可以帮助开发人员入手的资源（包括邮件列表，联机手册和Javadocs），但使用Apache HBase构建、设计和部署真正生产应用程序的信息相当有限。这正是写作本书的原因。

本书的目的是为了真正生产应用的HBase部署。虽然本书中讨论的每个用例已经部署并投入生产，但这并不意味着没有改进的余地，或者即使你不需要修改你的特定任务，但它确实展示了事情的实际情况。

这本书还包括了非常有用的故障处理内容（第三部分）。我们的目标是帮助你避免常见的部署错误。第三部分还提供了经常被我们忽略的性能调优内容，如垃圾回收和区域分配。

本书的读者对象？

本书主要针对那些架构师及开发人员而设计，希望他们能更好地理解大数据应用程序的部署。在这之前，你应该具备基本的Hadoop知识，包括所需组件的设置以及成

功安装过Hadoop集群，我们不会在Hadoop的配置或NodeManager功能上花费时间。阅读本书的架构师不需要有一个完整的Java知识，但必须充分了解部署章节的内容。这本书涵盖多个垂直用例，希望能够协助各个企业和初创公司。

架构师可以品读以细节为导向的用例章节，其中包括各个组件的部署方式以及它们是如何被集成在一起的。在开发章节，开发人员可快速查看详细代码示例，这样可以加快生产部署。部署章节提供了对特定API的深入了解以及相应的性能提升技巧，这样可大大减少我们的故障处理时间。那些对大数据好奇的人将会发现架构和部署章节都很有用，并且还可以深入了解HBase生态系统以及HBase的部署细节。

本书结构

本书分为三个部分：第一部分，HBase的介绍，涵盖的主题有：HBase是什么；HBase生态系统是什么样的以及如何部署它。第二部分，涵盖了具体用例，是本书的核心。我们希望这是你最常参考的部分，因为它包含了对你有用的提示和技巧。最后，第三部分讨论了故障处理，你应该经常参阅这一部分。我们希望这将是第二个最常参考的部分（以积极的态度，而不是被动的）。本部分提供了关于region数量控制、垃圾收集调优和避免热点等方面的内容。

额外的资源

这本书没有涵盖HBase的内部设计。我的好朋友Lars George通过《HBase权威指南》（O'Reilly出版）将HBase的内部设计带到了一个全新的水平。我们建议把他的书当作我们的启蒙书。这将有助于你更好地理解一些术语，而我们只是用一两段注释来解释它们。

虽然我们花了不少时间讨论部署HBase的细节，但本书不会涵盖部署HBase的大部分理论。Nick Dimiduk和Amandeep Khurana的《HBase in Action》（Manning出版）涵盖了部署HBase的实用性。虽然他们的书不太关注整个应用程序的开发，但在生产实践上做了较详细的介绍。

排版约定

本书使用了下述排版约定：

斜体（Italic）

表示新术语、URL、电子邮件地址、文件名和文件扩展名。

等宽字体（`Constant width`）

用于程序列表，以及用于引用程序元素的段落内，如变量或函数名、数据库、数据类型、环境变量、语句和关键字。

等宽粗体（**`Constant width Bold`**）

显示应由用户逐字键入的命令或其他文本。

等宽斜体（*`Constant width italic`*）

显示应由用户提供的值或由上下文确定的值替换的文本。

表示提示或建议。

表示一般注意事项。

表示提醒或警告。

使用代码示例

补充材料（代码示例，练习等）可从*https://github.com/ArchitectingHBase/examples*下载。

这本书是为了帮助你完成你的工作。一般来说，如果本书提供了代码实例，你可以在程序和文档中使用它。除非复制代码的重要部分，否则不需要与我们联系。例如，编写使用本书中多个代码块的程序不需要许可。出售或分发CD-ROM的例子来自O'Reilly的书籍需要许可。引用这本书来回答一个问题和引用示例代码不需要许可。将本书中的大量示例代码写入产品文档中，则需要获得许可。

我们推荐使用书属性的引用方式，但并不一定完全要求这样。书属性的引用方式通常包括标题，作者，出版商和ISBN。例如：“Architecting HBase Applications，Jean-Marc Spaggiari and Kevin O’Dell（O’Reilly），Copyright 2016 Jean-Marc Spaggiari and Kevin O’Dell，978-1-491-91581-3。”如果你觉得所使用的代码示例超出了上面的许可范围，可以随时通过邮件permissions@oreilly.com与我们联系。

Safari在线图书

Safari Books Online（*http://safaribooksonline.com*）是应需而变的数字图书馆，它同时以图书和视频的形式出版世界顶级技术和商务作家的专业作品。

Safari Books Online是技术专家、软件开发人员、Web设计师、商务人士和创意人士开展调研、解决问题、学习和认证培训的第一手资料来源。

Safari Books Online为企业、政府部门、教育机构和个人提供了多种套餐和价格。

订阅者可以在一个完全可搜索的全文数据库中访问上千种图书、培训视频和正式出版之前的书稿。这些内容由以下出版社提供：O’Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology等。关于Safari Books Online的更多信息，请访问我们的网站：*http://www.safaribooksonline.com*。

如何联系我们

请将有关本书的意见和问题反馈给出版商：

美国：

O’Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472

中国：

北京市西城区西直门南大街2号成铭大厦C座807室（100035）
奥莱利技术咨询（北京）有限公司

本书有一个Web页面，我们列出了勘误表、示例和其他相关内容信息。可以通过*http://bit.ly/architecting-hbase-applications*访问此页面。

要评论本书或咨询技术问题，请发送电子邮件至*bookquestions @oreilly.com*。

想了解更多关于我们的书籍，课程，会议和新闻的信息，请访问我们的网站*http://www.oreilly.com*。

在Facebook上找到我们：*http://facebook.com/oreilly*

在Twitter上关注我们：*http://twitter.com/oreillymedia*

在YouTube上观看我们：*http://www.youtube.com/oreillymedia*

致谢

Kevin和Jean Marc要感谢每一位让这本书成功出版的人，感谢他们所有的辛苦工作：我们的编辑Marie Beaugureau；在O'Reilly Media的卓越员工；撰写序的Lars Hofhansl；我们的主要审稿人Nate Neff、Suzanne McIntosh、Jeff "Jeffrey" Holoman、Prateek Rungta、Jon Hsieh、holoman、Sean Busbey和Nicolae Popa；以及给予我们无限支持和指导的各位作者：Ben Spivey、Joey Echeverria、Ted Malaska、Gwen Shapira、Lars George、Eric Sammer、Amandeep Khurana、Tom White。我们还要感谢Linden Hillenbrand、Eric Driscoll、Ron Beck、Paul Beduhn、Matt Jackson、Ryan Blue、Aaron "ATM" Meyers、Dave Shuman、Ryan Bosshart（谢谢你的鼎力相助）、Jean-Daniel "JD" Cryans、St. Ack、Elliot Clark、Harsh J Chouraria、AmyO'Connor、Patrick Angeles和Alex Moundalexis。最后，我们要感谢Cloudera和Rocana的所有人对我们的支持、建议和鼓励。

来自Kevin

感谢我最好的朋友和兄弟的全力支持和鼓励：Matthew "Kabuki" Langrehr、Scott Hopkins、Paul Bernier、Zack Myers、Matthew Ring、Brian Clay、Chris Holt、Cole Sillivant、Viktor "Shrek" Skowronek、Kyle Prawdzik和船长Matt Jones。我还要感谢我的朋友、同事、合作伙伴以及一路帮助过我的客户：Ron Kent、John Lynch、Brian Burton、Mark Schnegelberger、David Hackett、Sekou McKissick、Scott Burkey、David Rahm、Steve Williams、Nick Preztak、Steve "Totty" Totman、Brock Noland、Josh "Nooga" Patterson、Shawn Dolley、Stephen Fritz、Richard

Saltzer、Ryan P和Sam Heywood。特别感谢所有发布过他们用例以及提供咨询服务的人：Kathleen DeValk、Kevin Farmer、Raheem Daya、Tomas Mazukna、Chris Ingrassia、Kevin Sommer和Jeremy Ulstad。

特别感谢Mike Olson和 Angus Klein招聘我到Cloudera工作；感谢Eric Sammer、Omer Trajman和Marc“Boat Ready”Degenkolb 把我带到Rocana；感谢DonBrown的非议；最后一个感谢你，Jean-Marc，非常感谢你让我成为你的合著者。

来自Jean-Marc

感谢在写作本书的这段旅程中支持过我的所有的人。

第一部分

HBase概述

欢迎来到HBase应用架构的第一部分。在我们深入探讨架构设计和部署生产用例之前，先来梳理HBase的背景是很重要的。对于阅读过《HBase权威指南（O'Reilly）》或《HBase实战（Manning）》的人而言，第一部分将是一次深度复习。如果你阅读这些书籍已有一段时间，亦或是本书是你的HBase启蒙书，我们都建议阅读第一部分。

首先，我们将对HBase一般原理做一个高层次的概述。接下来，我们将检视HBase周围的生态系统。本书的目标并非面面俱到地介绍和HBase相关的每项技术，但是会给你一个HBase相关选项及整体知识点的理解。继生态系统后，我们将介绍和HBase相关的更深奥的主题：常见的HBase集群规模规划与调优。这样做的目的是帮你提前避免节点规模估算不当导致的错误，同时提供最佳的调优方法，将性能问题防患于未然。最后，我们将总结HBase部署概述。这将引导你按照书中的例子独自建立你的HBase单机实例。

第1章

什么是HBase

早在20世纪90年代，谷歌就已开始对网页创建索引，但很快就面临一些挑战。

第一个挑战和数据容量相关：网页迅速增长，从数千万直到如今的数十亿级别。随着时间的推移，构建网页索引也变得越来越难。

这导致了谷歌文件系统（GFS）的诞生并在其内部使用，在2006年该公司发布“Bigtable: A Distributed Storage System for Structured Data”GFS的白皮书。开源社区发现了这一契机，并在 Apache Lucene搜索项目中开始实施一套与GFS类似的文件系统：Hadoop。作为Apache Lucene项目的一部分，经过几个月的发展，Hadoop成为了Apache的自家项目。

伴随着谷歌开始存储越来越多的数据，很快它面临着另一个挑战。这一次和大规模的数据索引相关。如何在跨多个节点中存储一个巨大索引，同时保持高度的一致性，以及故障转移和低延迟的随机读取和随机写入，谷歌创建了一个被称为Bigtable的内部项目来满足需求。

再次，Apache开源社区又看到了利用Bigtable白皮书的一个绝好机会，开始了HBase的实现。最初的时候，Apache HBase是作为Hadoop项目的一部分开始的。

然后，在2010年5月，HBase毕业成为Apache自己的顶级项目。项目创建多年后的今天，Apache HBase项目继续蓬勃发展和壮大。

正如Apache HBase网站所描述的那样，HBase“是Hadoop的数据库，它是一个分布式、可扩展的大数据存储数据库”。如果你有丰富的数据库经验，这种简洁的描述可能误导你。更准确地说，它是一个列存储，而不是一个数据库。

这本书将有助于你很快在大脑里形成明确的认知。

更具体地说，HBase是一个基于Java、开源、NoSQL、非关系型、面向列的、构建于Hadoop分布式文件系统（HDFS）上的、仿照谷歌的BigTable的论文开发的分布式数据库。HBase给Hadoop 生态系统引入了Bigtable的大部分功能。

HBase是要建立一个可容错并托管一些大的数据稀疏表（亿元/兆行数以百万计列）的应用，同时允许非常低的延迟和近实时的随机读取和随机写入。

HBase的设计保证了以一致性为前提的可用性，并且由于能够快速自动完成故障转移，因此也具有高可用性。

HBase还提供了很多功能，我们将在本书的后面进行介绍，包括：

- 副本数。
- Java、REST、Avro和Thrift API。
- 基于HBase 数据框架的MapReduce。
- 表自动切片。
- 负载均衡。
- 压缩。
- 布隆过滤器。
- 服务器端进程 (过滤器和协处理器)。

HBase的另一方面是允许创建和使用灵活的数据模型。HBase不强迫用户有一个强大的模型列的定义，它可以在线根据需求创建。

除了提供原子及强一致的行级操作，HBase还对整个数据集实现一致性和分区容忍性。

然而，你也需要意识到HBase的局限性：

- HBase不是一个SQL数据库替代品。
- HBase不是事务性数据库。
- HBase不提供SQL API。

面向列与面向行

如前所述，HBase是一个面向列的数据库，这大大不同于传统的面向行的关系数据库管理系统（RDBMS）。这种差异极大地影响了文件系统中存储以及检索数据的方式。在面向列的数据库中，系统将数据表存储为稀疏的数据列，而不是作为整个行数据进行存储。HBase选择列模式的存储方式，这种方式将允许新一代的用例和数据集能够快速部署和迭代。传统的关系模型要求数据结构统一，并不能满足社会媒体、制造业和Web数据的需求。本质上，这种数据往往是稀疏的，这意味着并不是所有的行都是相等的。由于具备快速存储和访问稀疏数据的能力，HBase支持在列数目为100的行后面接着存储列数目为1000的行，且这些不同列数目的行并不会互相影响。HBase的数据格式还可以定义松散的表。HBase创建表的时候，只有表名称和列族数量是必要的。这使得在写入过程中能够动态分配，当处理非静态和不断变化的数据的时候，这是非常有价值的。

利用面向列的模式会影响应用程序和用例的多个方面设计。未能正确地认识HBase的局限会导致HBase特定的操作性能下降，包括读/写性能、检查和交换（CAS）的操作。当讲解到如何正确利用HBase API和成功部署架构设计时，我们会提到这些细微差别。

实现和使用示例

目前HBase部署在全球数以千计的不同规模的企业中。在这本书中一一罗列出它们是不可能的。当你开始或完善你的HBase的历程时，考虑下面的大规模公共HBase实现：[注1]

- Facebook消息平台。
- Yahoo!。
- eBay。

在第二部分中，我们将集中关注目前在生产环境中的四个实际使用示例：

- 使用HBase作为Solr的底层引擎。

注1：对于这些实现的深入讨论，请分别参阅“The Underlying Technology of Messages”，“Apache HBase at Yahoo! – Multi-Tenancy at the Helm Again”，“HBase: The Use Case in eBay Cassini”。

- 使用HBase处理实时事件。
- 使用HBase作为主数据管理（MDM）系统。
- 使用HBase作为文件存储的替代品。

随着时间的推移，HBase不断发展，其Logo也一样。如今，HBase的Logo采用了一种简洁且新潮的文本表现形式。由于Hadoop 生态系统的其他所有项目都有吉祥物，HBase最近投票选择虎鲸作为其吉祥物（见图1-1）。

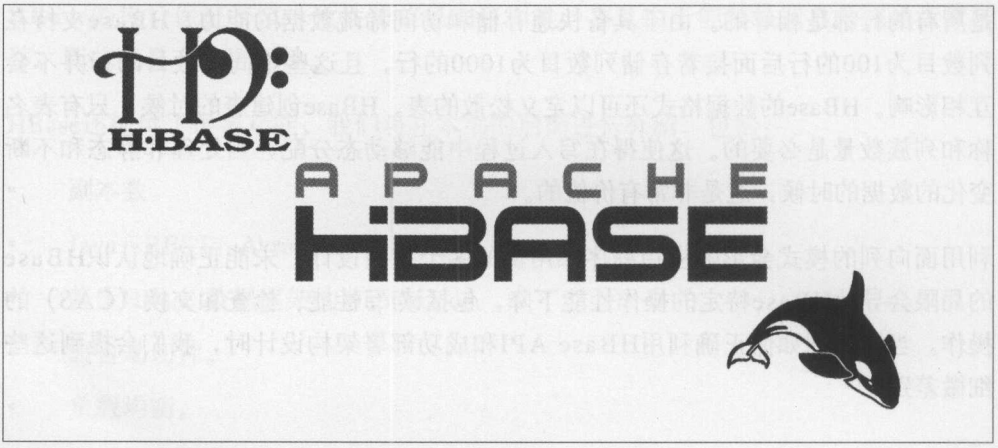


图1-1：HBase从过去到现在的Logo，包括吉祥物

第2章

HBase原理

在上一章我们大致了解HBase的基础知识，接下来，我们将深入细节，更多地了解HBase的原理和内部构造。尽管为了构建好的设计而去理解HBase的原理是非常重要的，但也不代表你必须知道内部所有的细节。在本章，我们将重点讨论HBase的内部构造，这将有助于你达到更好的设计效果。

表格式列族

和传统数据库一样，HBase将数据存储在表中，同时也有行键和列名称的概念。和传统的数据库不同的地方在于它引进了列族的概念，我们稍后将介绍。但是与RDBMS相比，尽管HBase命名与之相同，但是表和列的工作方式不一样。如果你习惯了传统的关系数据库，你将会对这里大多数名词很熟悉，不过由于具体的实现方式是不一样的，所以在你学习HBase的时候，需要暂且搁置你已认知的数据库知识，并将先入为主的观念放在一边。

在HBase中，你将会发现两种不同类型的表：系统表和用户表。系统表由HBase内部使用来追踪元数据信息，例如表的访问权限列表（ACL）、表和区域的元数据信息、命名空间等。你没有必要去查看这些表。用户表是为你的用例创建的表。除非你在创建的时候特别指定其命名空间，默认情况下创建的表将属于default命名空间。

表排版

一个HBase表由一个或者多个列族（CF）组成，一个列族又包含很多列（我们称列为列限定符，简称CQ），每列存储相应的值。和传统的关系型数据库不一样，

HBase表是稀疏表，一些列可能根本不存储值。在这样的情况下也不会存储null值。而且该列将不会被添加到表中。一旦一个row key及相应的列值生成了，将会被存储到表中。

在HBase 中，人们使用很多单词来描述不同的部分：行、列、键、单元格、值、行键、时间戳等。为了确保我们说的是同一个事情，我们将对HBase的术语进行统一，一行由很多列组成，全部由相同的键引用。一个特定的列和一个键称为单元格。一个单元格可以有很多版本，由不同时间戳的版本来区分。单元格可以叫做键值对。因此，一行由一个键引用，每行由一组单元格组成，其中每一个单元格又由指定的行名和特定列名确定。

有值的列才会被存储到底层的文件系统。另外，即使在创建表时需要定义列族，也不需要提前定义列名称。当列族插入数据时，列名称可以动态生成。因此，在不同的行中会有可能存在数百万动态创建的列名。

为了更快地查询，键和列会按照字母排序存储在表中，同时也存储在内存中。

HBase会根据字节值将key进行排序，所以“AA”将会在“BB”之前。如果你把数字按照字符链存储，那么“1234”将会在“9”前面。如果你必须存储数字，那么为了节省空间并保持顺序，你需要存储它们的字节表示。在这个模型中，整数“1234”将以0x00 0x00 0x04 0xD2存储，而“9”将会以 0x00 0x00 0x000x09存储序。对这两个值进行排序后，我们可以看到，0x00 0x00 0x00 0x09存储在 0x00 0x00 0x04 0xD2之前。

了解HBase表结构的最简单的方法就是直接看表。图2-1展示了一些表的标准数据表示，其中有些行中的列被填充了值，有些列却没有。

Keys	CQ1	CQ2	CQ3	CQ1	CQ2
042	C		E		
123	I	A		D	E
999	B	H			G

图2-1：HBase表的逻辑表示

因为它们将在文件系统中被用于创建文件和目录，表名称和列族名称需要使用可打印字符。此限制不适用列，但如果你使用的是外部应用程序显示表的内容，则使用只可打印字符来定义它们会更加安全。

表存储

表存储有多个方面。第一个方面是HBase如何将单独的一个值存储至表中。第二个方面涉及如何将所有这些单元格存储在一起，以形成一个表。

如图2-2所示，从外到内，一个表是由一个或者多个region组成，一个region由一个或者多个列族组成，一个列族由一个store组成，一个store由唯一的memstore 加上一个或者多个HFile组成，HFlie又是由block组成，而block是由cell组成。

RegionServer上的memstore的数目为region的数目和正在接收写入数据的列族数目的乘积，且该RegionServer上所有的memstore都共享同一个块预留内存。

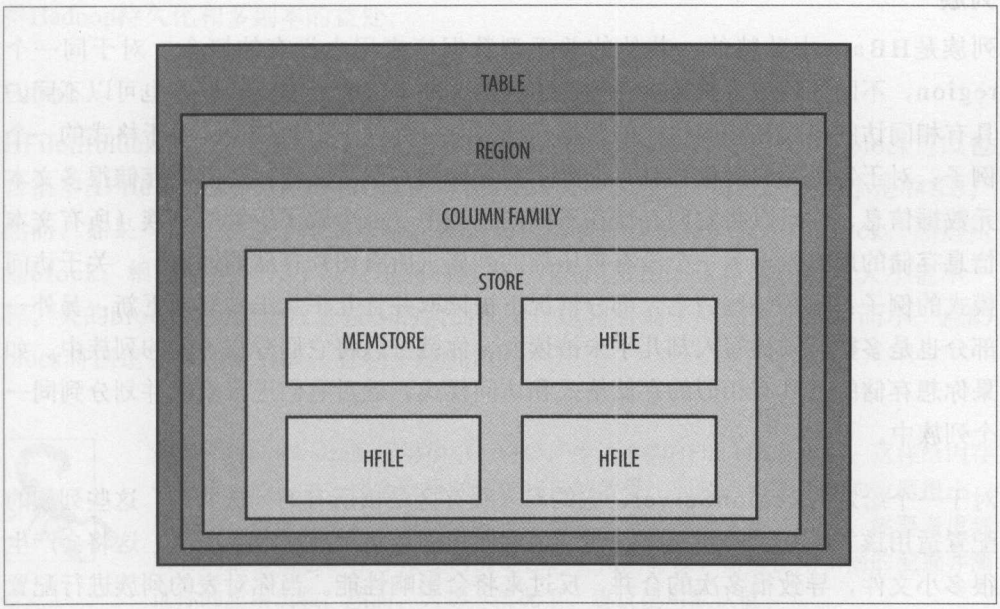


图2-2：存储层

下面是对HBase所有这些不同逻辑层的概述。

region

所有的行以及相关的列一起形成了一张表。但是，为了提供可扩展以及快速随机访问的功能，HBase不得不将数据分布在多个服务器中。为了达到这个目的，表被分割成多个region存储，每个region将会存储一个指定区间的数据。region将会被分配到RegionServer以服务于每个region的内容。当新的region被创建后，过了配置的那段时间，HBase的负载平衡器将会把数据移动到其他的RegionServer上，以确保HBase集群负载均衡。类似预分区，还有很多有效的关于region优化的策略。相关的知识点将会在下面几章陈述。

每个region都有一个起始键和一个结束键来定义它的边界。所有这些信息将随着文件保存在region中，也会保存在hbase：meta表中（对于HBase 0.96之前的版本则保存在.META.中）。通过这张表能够跟踪所有的region信息。当它们变得太大，region可以分裂。如果需要，region也可以合并。

列族

列族是HBase中独特的、其他的关系型数据库应用中没有的概念。对于同一个region，不同的列族会将数据存储在不同的文件中，而且它们的配置也可以不同。具有相同访问模式和相同格式的数据应该被划分到同一个列族中。关于格式的一个例子：对于每个用户画像信息，你除了要存储用户图片文件，还需要存储很多文本元数据信息，你可以将它们存储在不同的列族中：一个做了压缩的列族（所有文本信息存储的地方），另一个没有被压缩的列族（所有图片存储的地方）。关于访问模式的例子：如果一些信息大部分情况下被读取并且几乎从未被写入更新；另外一部分也是多数时候被写入却几乎未被读取，你就可以将它们分在不同的列族中。如果你想存储的列具有相似的存储格式和访问模式，就对它们进行重组并划分到同一个列族中。

对于一个给定的RegionServer，它的写入缓存区是由所有的列族共享，这些列族的配置适用该主机托管的所有region。过度使用列族将对缓存产生压力，这将会产生很多小文件，导致很多次的合并，反过来将会影响性能。当你对表的列族进行配置时，技术上并没有数目的限制。但是，综观过去的几年，我们工作中遇到的绝大部分的用户案例都是需要仅要求一个列族。有时有些需要两个列族，但是每次我们看到超过两个列族时，通常都是建议其减少列族的个数来改善效率。如果你的设计包括三个以上的列族，你可能需要认真考虑下是否真的需要这么多列族，大部分情况下它们可以重新分组。如果你没有对两个列族之间做一致性约束，而且数据也将在不同的时间存储到各自的列族中，你可以创建两个表，每个表都有一个列族，而不

是给一个表创建两个列族。这个策略在决定region的大小的时候很有效。实际上，虽然保持两个列族大小相同更好。通过把它们分割成两个不同的表，这样使其更容易独自增长。

第15章为列族提供了更多的细节。

Store

我们发现一个store对应着一个列族。一个store对象由一个memstore和零个或更多的strore File(称为HFile) 组成。store是存储所有写入表中的信息的实体，并且当数据需要从表中读取时也将被用到。

HFile

当内存写满必须要刷新到磁盘的时候，HFlie就会被创建。随着时间的推移，HFlie最终会被压缩成大的文件。它们是HBase用来存储表数据的文件格式。HFlie由不同种类的block块组成(如索引块和数据块)。HFlie存储在HDFS上，因此它们也能够获得Hadoop持久化和多副本的益处。

Block

HFile由block组成。这些block不应该和HDFS的block混淆。一个HDFS block可以包含很多HFile block。HFile block通常在8KB和1MB之间，但是默认大小是64KB，然而，如果一个表配置了压缩选项，HBase仍然会产生64KB 大小的block，然后压缩block。根据数据的大小以及压缩的格式，压缩后的block存储在磁盘大小也不一样。大的block将会创建数量较少的索引数据，这将有助于顺序表访问，而小一点的block将创建更多的索引值，有利于随机访问。

如果你将block size配置得很小，将会产生过多的HFlie block 索引，这样给内存带来很大的压力，将会取得和预期相反的效果。同时，由于压缩的数据很小，压缩率也很低，数据容量将会增大。当你决定修改默认值的时候，需要考虑到所有的细节信息。在你做任何决定性的变化的时候，需要使用不同的配置并测试你的应用负载。然而，大部分情况下，建议使用默认值。

下面主要的块类型会在HFile文件涉及（因为它们大多是内部的细节实现，我们只会提供一个概述；如果想对具体的块类型了解更多，参考HBase的源代码）：

数据块

一个数据块将包括压缩了或未被压缩的数据，但不是两者的组合。数据块包括删除和插入标记。

索引块

当查询指定的行时，HBase将会使用索引块来快速跳转到相应的HFlie地址。

布隆过滤块

这些数据块是用来存储数据块索引相关信息的。当查询指定key，使用布隆过滤块索引来跳过文件解析。

Trailer块

Trailer块包含了文件的其他可变大小的部分的偏移量，它也包含HFile版本信息。

数据块采用逆序存储。这就意味着，数据块也按照逆序写入，而不是先在文件开头放入索引后将其他数据写入。先存储数据块然后存储数据块索引，Trailer数据块存储在最后。

单元格

HBase是面向列存储的数据库。这就意味着每一列将单独存储，而不是单独存储整个行。因为数据值能在不同的时间插入，因此在HDFS上最终变成不同的文件。

图2-3展示了HBase如何从图2-1存储数据。

042	CF1	CQ1	C
042	CF1	CQ3	E
123	CF1	CQ1	I
123	CF1	CQ2	A
123	CF2	CQ1	D
123	CF2	CQ2	E
999	CF1	CQ1	B
999	CF1	CQ2	H
999	CF2	CQ2	G

图2-3：HBase物理表

如你所见，只有带有值的列才会被存储，没有值的列将不会被存储。

每一行将会在内部以指定的方式存储。图2-4展示了单个HBase单元格存储的格式。

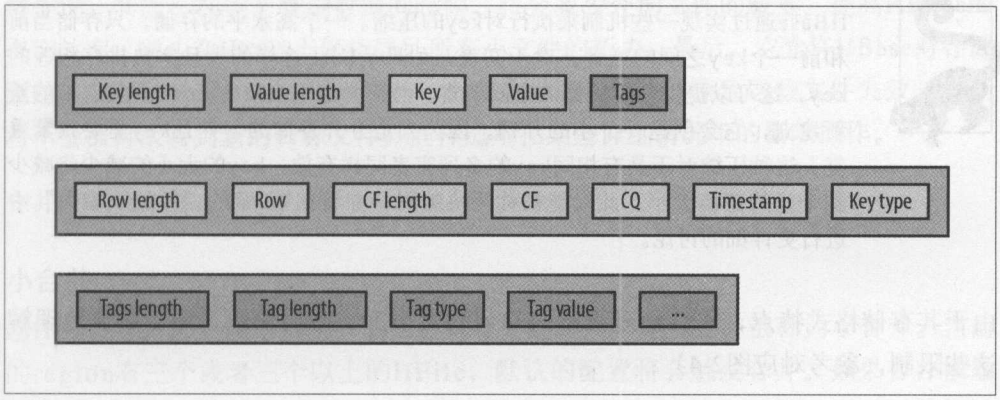


图2-4：HBase单元格格式

图2-5展示了图2-3的第一个单元格如何被HBase存储。只有当HFlies版本为3的时候，数据的标签是可选的。如果一个单元格没有任何标签，那么它也将不会被存储。

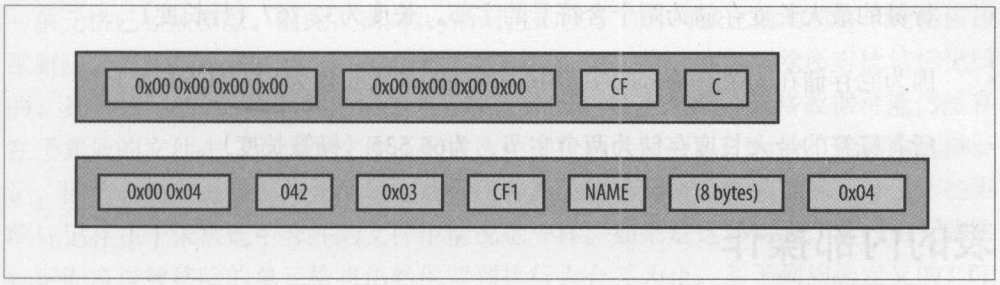


图2-5：单元格的例子

“key type”字段代表不同的可能的HBase操作如下：

- `Put`
- `Delete`
- `DeleteFamilyVersion`
- `DeleteColumn`
- `DeleteFamily`

HBase存储数据的方式极大影响了HBase的表设计。从前面的例子我们可以看出，为

了存储1字节的数据，HBase必须增加额外的31字节。这种开销根据CQ名称、CF名称和其他一些标准而有所不同，但仍然非常重要。

HBase通过实现一些机制来执行对key的压缩。一个高水平的存储，只存储当前和前一个key之间的增量。对于高表，如果是URL这样的大且字典排序相近的key，这可以提供一些良好的空间节省。然而，由于当前的key需要从之前的重新建立，它会创建一个小的开销。因为列是分开存储的，并且key需要频繁重复，这种压缩对于具有相同key的多列宽表同样有效。key的大小的减少将减少整个行的整体大小。此功能被称为块编码，并将在第7章的“数据块编码”中进行更详细的讨论。

由于其存储格式特点，HBase对于不同的列的大小有一些局限性（为了简单地理解这些限制，参考对应图2-4）：

- 行键加上列族以及列限定符是以4字节大小存储，它们的最大长度为2^31–1–12或者2 147 483 635 (key 长度)。
- 以四字节存储的值的最大长度为2^31–1–12或2 147 483 635（值长度）。
- 行键的最大长度存储为两个含符号的字节，长度为32 767（行长度）。
- 因为它存储在一个有符号的字节中，列族的最大长度为127（CF长度）。
- 所有标签的最大长度存储为两个字节，为65 535（标签长度）。

表的内部操作

HBase的可扩展性是基于其对数据的重组以合并成更大的文件，然后将表里面的数据分散到很多服务器上的能力。为了达到这个目标，HBase具有三大机制：合并、分裂、重平衡。这三个机制对用户是透明的。但是如果设计不好或者使用不当，将可能影响服务器的性能表现。因此，通过了解这些机制将有助于了解服务器的反应。

合并

HBase将所有接收到的操作保存到memstore内存区。当内存缓冲区满了之后，会将数据刷新到磁盘（参见第15章查看memstore以及其刷新机制的详细信息）。随着时间的推移，这样的操作会不停地在HDFS上创建很多小文件，并且根据我们稍后会

提到的具体标准，HBase将会选择相应的文件组合成更大的文件。这会在多个方面让HBase受益。首先，新的文件将由托管的RegionServer进行写入，保证数据存储在HDFS本地磁盘上。本地写操作将允许RegionServer在本地查询文件，而不是通过网络查询。其次，在用户查询数据的时候，将会减少查询文件的数量。提高HBase的查询效率，减少在HDFS上保持寻址所有小文件的压力。最后，它允许HBase对存储到这些文件的数据做一些清理工作。如果生存周期(TTL)导致这些文件失效，它们将不会被再次写到新的目标文件。同样适用在某些详细情况下的删除操作。

合并类型有两种：

小合并

当HBase选择部分而不是所有的来进行合并的时候，这种合并被称为小合并。当前的region有三个或者三个以上的HFile，默认的配置将会触发合并。如果合并被触发，HBase将会基于合并规则选择一些文件进行合并。如果store中的所有文件都被选中，那么这个合并将会演变成大合并。

小合并对数据进行一些清理操作，但不是所有的信息都会被清理。当你对一个单元格执行删除操作的时候，HBase将会存储一个标记，来表示所有比该单元格旧的同一单元格已经被删除。因此，相同key所对应的位于该时间戳之前的所有单元格应该被删除。当HBase在执行合并操作时发现删除标记，将会确保删除所有比该标记旧的，具有相同的key和列限定词的单元格。然而，由于一些单元格数据可能仍然存在于其他的文件中，而这些文件还未被选中进行合并，所以HBase不能删除这些标记，因为它仍然适用，并无法确定是否还有其他的单元格需要被删除。对于那些删除标记存在于未被选中合并的文件中情况也一样。如果是这种情况，那些由于删除标记而应该被移除的单元格将仍然保留到执行大合并为止。基于列族级定义的TTL的过期单元格将被删除，因为它们不依赖其他未被选中的文件的内容，除非该表已被配置成保持最少的版本数。

理解单元格的版本总数和合并的关系很重要。当决定需要保留数据的版本数后，最好就是将该数字视为在给定时间内的最小版本数。一个很好的例子就是只有单个列族的表，该表被配置为保留最大为3的版本数。只有两种情况下HBase将会删除多余的版本。第一种是刷新数据到磁盘的时候，第二种是执行大合并的时候。返回给客户端的单元格版本的个数，通常是根据表的配置而定，然而，当使用`RAW=>true`的时候，你可以获得所有版本的数据，下面深入探究下几种情况：

- 执行四次put，并不刷到磁盘，紧接着执行scan，不论版本数，将会返回四个版本的数据。
- 执行四次put，刷到磁盘，紧接着执行scan，将会返回三个版本的数据。
- 执行四次put并刷到磁盘，再执行四次put，并刷到磁盘，紧接着执行scan，将会返回六个版本的数据。
- 执行四次put并刷到磁盘，再执行四次put并刷新到磁盘，再进行大合并，紧接着执行scan，将会返回三个版本的数据。

大合并

当所有的文件被选中进行合并的时候，我们称为大合并。大合并和小合并工作原理类似，除了前者会将应用于所有相关单元格的delete market删除，此外同一个单元格所有多余的版本也将被丢弃。大合并可以对指定的region的列族级别或在region级别或者表级别进行手动触发。HBase也能够配置，以便于周期性地执行大合并。

自动每周一次的合并能够在任何情况下发生，这个时间取决于你的集群启动时间。这就意味着，当你几乎没有HBase的通信量时，自动合并可以突然发生，但是这也意味着，它可以精确的发生在峰值活动正在进行的时候。因为合并需要读取和重写所有的数据，大合并是I/O密集型操作，将会对你的集群反应时间和SLA产生很大影响。因此，强烈建议完全关闭自动合并，并且在当你知道在对集群影响最小的时候，你自己启动定时任务来触发合并操作。我们也建议不要同时合并所有的表。与将所有表的操作放在一周的同一天执行相反，可以将合并操作分散到整个周。最后，如果你的集群真的含有很多表和分区，建议开发一个程序检查每个分区的文件个数以及最旧文件的生命周期；在region级别上，只要超过你设定的文件数或最旧的文件的生命周期超过你预先配置的期限(即使只有一个文件)都可以触发一个合并操作。这样有助于保持region数据的locality值，同时降低你集群的I/O。

分裂（自动分片）

分裂操作是合并的相反操作。当HBase将多个文件合并在一起的时候，如果在合并过程中没有太多的值被丢弃，它将创建一个更大的文件。输入的文件越大，就需要更多的时间去解析合并它们等。因此，HBase试图将它们保持在可配置的最大值之下。在HBase0.94这个版本以及更旧的版本中，默认值是1GB，然后在后面的版本中，这个值增加到10GB。当其中一个region的列族达到10GB之后，为了优化负载均衡，HBase将会对指定的region触发拆分机制，将region分裂成两个新region。因为region边界适用给定region的所有列族，所有的列族将会按照同样的方式进行分裂，

即使它们远小于配置的最大值。当一个region分裂后，将会分割成两个新的较小的region，第一个region的start key为原来分区的start key，第二个region的end key是原来分区的end key。第一个region的end key以及第二个region的start key由HBase决定，它将会选出最优中点。HBase将会尽量选择中间点，然而，我们不想这个过程耗费太多的时间，所以它不会在一个HFile块内分裂。

关于拆分有些事情需要注意。首先HBase从不会在同一行的两列之间进行分裂。所有的列将会被保存在同一个region里面。因此，如果你有很多列或者非常大的列，单个行的值可能比配置的最大值都大，而HBase不能分裂它。你要避免这种整个region只服务于一行的情况。

同时，你还需要记住HBase将会拆分所有列族。即使你的第一个列族达到10GB的阈值，而第二个列族却只有少数几行或者几千字节，这两列族都会被拆分。分裂后的第二个列族将会在所有的region里面分布着微小的文件。这不是你想看到的状态，同时，你也想重新检查你的表设计来避免这样的事情。当你遇到这样的情况，并且在你的两个列族之间也没有很强的一致性需求时，可以考虑将它们分裂成两个表。

最后，不要忘了分裂表是需要付出代价的。当一个region分裂并均衡后，region数据在本地的百分比将会降低，直到下次合并。这会影响读取性能，因为客户端会到达托管着region的RegionServer去获得数据，然而当region做完均衡之后，将需要通过网络从其他的RegionServer上获得数据以服务请求。同时，更多的region将会对master、HBase：meta表，以及region服务产生更大的压力。在HBase的世界，region分裂是合理并正常的，但是你还需要关注它们。

图2-6展示了一个含两个列族的表在分裂之前和分裂之后的状态。如你所见，一个列族明显比另外一个大很多。

均衡

Region分裂后，服务器有可能宕机，新的服务器可能加入到集群中，因此，在某种程度上，数据将不会很合理地分布在你所有的RegionServer上。为了帮助集群保持合理的分布数据，每5分钟（默认配置的调度时间）HBase Master将会运行一个负载均衡器来保证所有的RegionServer管理和服务着近乎相同数目的region。

HBase有几种不同的负载均衡算法。0.94版本之前，HBase使用的都是`SimpleLoadBalancer`，但是从0.96版本之后开始使用`StochasticLoadBalancer`。尽管推荐使用默认配置的负载均衡器，但你也可以开发自己的负载均衡器并要求HBase使用它。

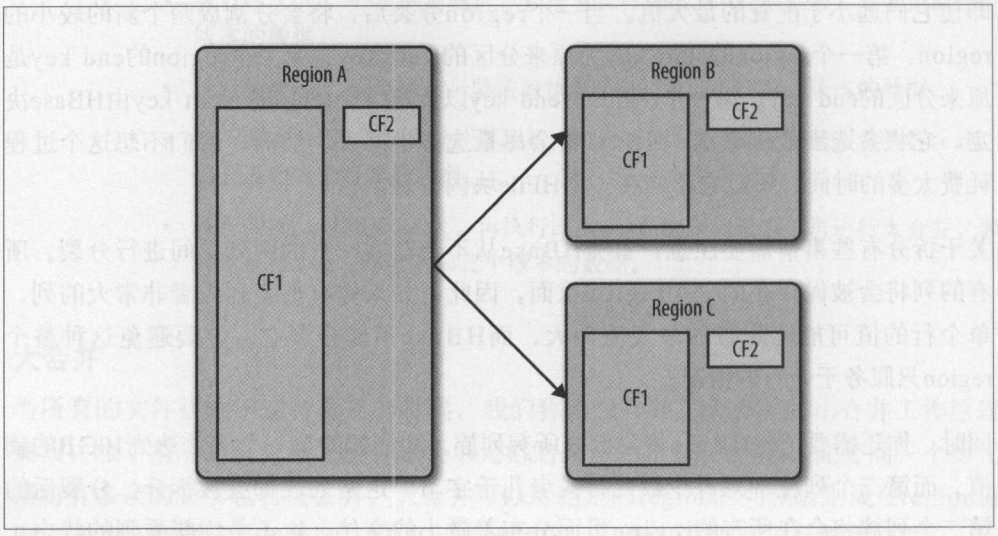


图2-6：两个列族的表分裂之前和分裂之后

当一个region被负载均衡器从一个服务器移动到另外一个新的服务器时，在几毫秒内该region将会不可用，同时丢弃它本地的数据，直到下一次做大合并操作的时候。

图2-7展示的是master如何将region从负载最重的服务器分配到负载压力小点的服务器。超负荷的服务器接受来自master的命令将region关闭并转移到目标服务器。

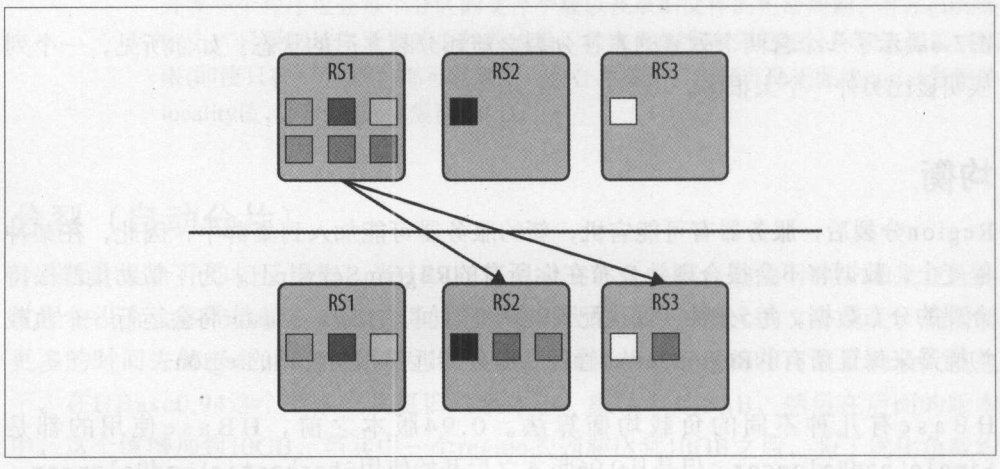


图2-7：超负荷的RegionServer迁移示例

依赖

HBase仅仅需要其他一些服务以及应用就能够运行。和HDFS一样，HBase用Java语言开发，需要最新的JDK来运行。正如我们看见的HBase依赖HDFS。然而，之前已经做过一些工作使得它能够在一些其他的文件系统上运行，如亚马逊S3。

HBase需要将HDFS和HBase运行在相同的节点上。这并不意味着HBase需要运行在所有的HDFS节点上，但我们仍然强烈建议这样做，因为这可能也会导致负载很不均衡的情况。HBase同时也依赖zookeeper来监控其服务器的健康状态，提供高可用的性能以及追踪备份的进程、正处于激活状态的Mastrer、存在的表的列表等信息。

HBase2.0正在努力减少对zookeeper的依赖。

HBase 角色

HBase有两个主要的角色：master（有时也被称为HBase Master、HMaster或者HM）和RegionServer(RS)。我们也可以使用Thrift和REST服务器通过不同的API来获得HBase数据。

图2-8展示了如何在不同类型的服务器上布置不同的服务。最近的HDFS版本允许超过两个NameNode。它允许所有的master 服务器运行一致性的服务（HMaster、NameNode和 ZooKeeper）。其中，运行的集群只有两个NameNode，而不是三个，这也是完全可以的。

Master 服务器

HBase的Master服务器是集群的大脑，负责下面这些操作：

- Region分配。
- 负载均衡。
- Regionserver恢复。
- Region分裂完成监控。
- 追踪处于活动和宕机状态的服务器。

为了达到高可用性，单个集群可以有多master。然而，每次只有一个master处于活动状态，负责上面的操作。

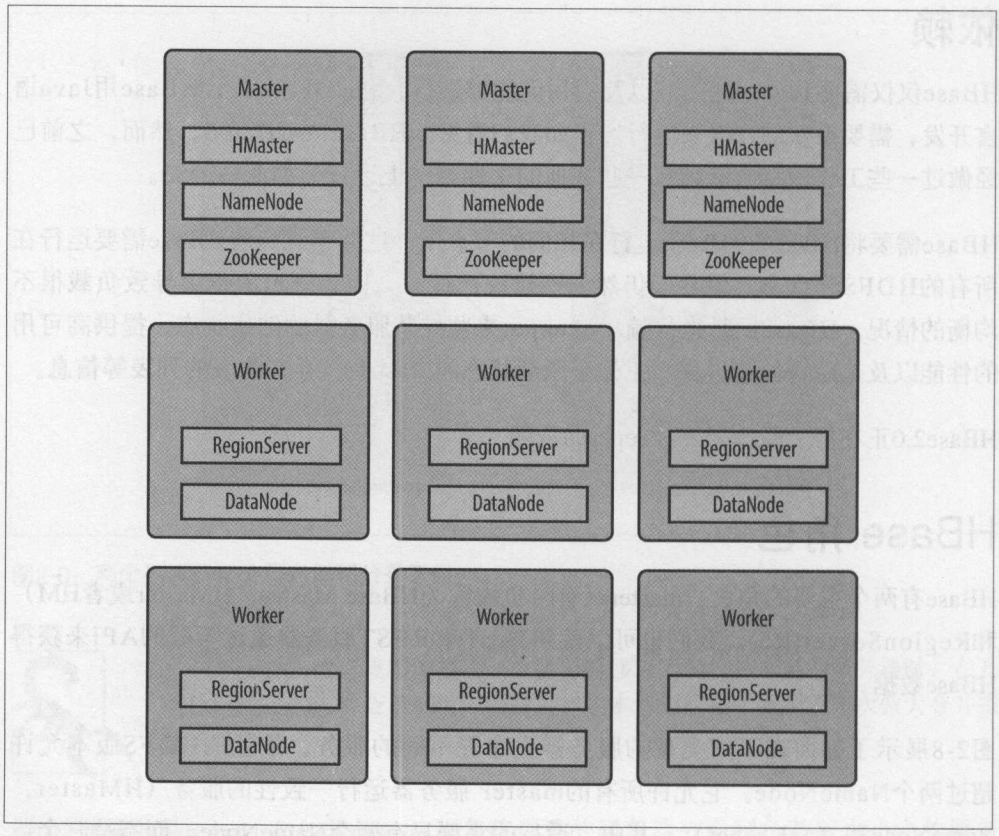


图2-8：一个理想的服务配置分布图

和HBase RegionServer不一样，HBase Master不会有很大的负载压力，可以被安装在内存和处理器核数比较小的服务器上。然而，作为集群的大脑，它必须提供更可靠的服务。如果集群中的两个RegionServer宕机后，集群还会继续运行，但是如果两个Master宕机，那么你的集群将会很危险。正因为如此，即使RegionServer在创建的时候，磁盘没有按照RAID配置或者双电源等，最好还是创建健壮性更强的HBase Master。强烈建议在带RAID驱动的操作系统以及双电源保障的硬件上创建HBase Master（其他如NameNodes、ZooKeeper等Master服务）。为了提高整个集群的稳定性，HBase 2.0正在做一些努力让Master监控一些zookeeper正在监控的信息。

只要没有RegionServer宕机或者region分裂，集群可以在没有Master的情况下运行。

RegionServer

一个RegionServer（RS）是托管并服务HBase region以及HBase数据的应用程序。当

用户请求数据的时候（到指定的region读/写数据），Java API的请求会直接发送到相应的RegionServer上。这是为了保证HBase Master或者活动的zookeeper服务不会成为过程中的瓶颈。

RegionServer决定并处理region的分裂和合并，同时将信息报告给Master。

尽管一个物理机上运行多个RegionServer在技术上是可行的，我们仍然建议在一个物理节点上运行一个RegionServer，并为其提供在两个服务器之间共享的资源。

当客户端第一次尝试从HBase读取数据的时候，首先，它会连接zookeeper寻找master服务器，并在HBase：meta定位出region的信息（它要查找的region的位置信息以及RegionServer信息）。若后面同样的客户端请求同样的region，所有这些连接zookeeper的过程都会被跳过，客户端直接跳到相关的RegionServer上获得数据。这就是为什么在可能的情况下，使用同一台机器执行多次操作的一个重要原因。

Thrift Server

Thrift Server可以被用作网关，允许使用其他语言开发的应用程序调用HBase服务。虽然Thrift Server可以使用Java调用HBase服务，不过还是建议直接使用Java API代替Thrift Server。HBase提供的Apache Thrift 模式可以让你使用你想用的语言来使用HBase。目前Thrift 模式共有两个版本。版本1是传统的模式，并保持对它的外部应用程序的兼容性。版本2是新版本，包括一个更新的模式。上述两个模式能够在下面的HBase代码路径找到。

Version 1 (legacy)

HBase-thrift/src/main/resources/org/apache/hadoop/HBase/thrift/HBase.thrift

Version 2

HBase-thrift/src/main/resources/org/apache/hadoop/HBase/thrift2/HBase.thrift

Java客户端能够连接任何一个RegionServer，但是C/C++客户端不可以，它只能和Thrift Server通信。这样会产生性能瓶颈，不过启动多个服务将会减少瓶颈点。

通过Thrift API的方式并不能保证所有的HBase API请求都会被Thrift API服务获取。Apache HBase社区试图尽可能保持它们都是最新的状态，但有时候会有一些报告丢失并被添加回来。 如果正在寻找的API调用在Thrift模式中不可用，请将其报告给社区。

REST Server

HBase同样也提供了REST Server API，通过该API，客户端以及管理性操作能够被执行。Rest API能够通过HTTP请求直接被客户端应用或者命令行应用（如curl）调用。通过指定HTTP头文件中的Accept字段，你可以让REST server来返回不同格式的结果。下面是相关格式：

- `text/plain` (consult the warning note at the end of this chapter for more information)
- `text/xml`
- `application/octet-stream`
- `application/x-protobuf`
- `application/protobuf`
- `application/json`

让我们看一个非常简单的创建表和填充表的示例：

```
create 't1', 'f1'
put 't1', 'r1', 'f1:c1', 'val1'
```

下面是一个例子，通过HBase REST API调用的方式，从XML的内容检索出我们已经插入的单元格数据。

```
curl -H "Accept: text/xml"http://localhost:8080/t1/r1/f1:c1
```

返回值如下：

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<CellSet>
  <Row key="cjE=">
<Cell column="ZjE6YzE=" timestamp="1435940848871">dmFsMQ==</Cell>
  </Row>
</CellSet>
```

被base64编码的值可以通过下面的命令解码：

```
$ echo "dmFsMQ==" | base64 -d
val1
```

如果不想解码XML和based64值，你可以使用`octet-stream`格式：

```
curl -H "Accept: application/octet-stream" http://localhost:8080/t1/r1/f1:c1
```

将会返回：

```
val1
```

虽然HBase代码引用了text/html格式，但它还没有实现，也不能使用。

因为返回结果不可能支持所有的格式，所以只有一些格式实现了一些调用。实际上，即使可以调用/*version*或/*t1*/*schema* URL来获取text/plain格式，但是会返回/t1/r1/f1:c1的失败信息。因此，在选择格式之前，请确认你需要的格式。

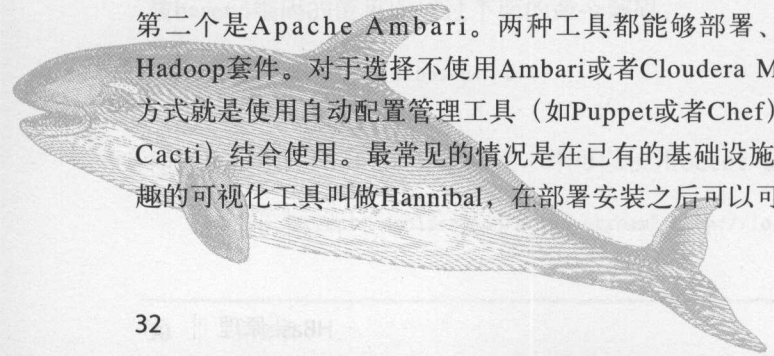
第3章

HBase生态系统

正如我们所知，HBase是作为Hadoop生态系统的一部分而设计出来的。好消息是，当创建新应用时，HBase本身就提供了一个强大的生态系统。这一点都不奇怪，因为HBase重要的功能就是服务生产数据并支持用户应用。HBase周边有很多工具，包括SQL查询层， ACID 事务一致性处理系统到管理系统以及客户端库。本书不会对数据库相关的每一个应用做深入讲解，因为每个话题本身都需要一本书来阐述。我们会回顾最杰出的工具并讨论它们的一些优点和缺点。接下来，我们将会看到顶级生态系统工具中最有意思的特征。

监控工具

通常，关于Hadoop和HBase最热门的话题之一就是管理和监控工具。多年来一直支持Hadoop和HBase，我们可以证明任何管理软件都比没有更好。说真的，想想你能想到的最坏的事情（就像淹死在一个黄色的芥末酱里），或者你最害怕的事情，然后把它成倍扩大，那就是在没有任何支持的情况下调试一个分布式系统。Hadoop和HBase都是以XML格式的文件来配置的，你可以手动创建它们。也就是说，在Hadoop生态系统中部署HBase集群有两个主要的工具。第一个是Clouera Manager，第二个是Apache Ambari。两种工具都能够部署、监控和管理各种公司的完整Hadoop套件。对于选择不使用Ambari或者Cloudera Manager安装情况，典型的部署方式就是使用自动配置管理工具（如Puppet或者Chef）和监控工具（如Ganglia或者Cacti）结合使用。最常见的情况是在已有的基础设施上使用这些工具。还有一个有趣的可视化工具叫做Hannibal，在部署安装之后可以可视化HBase的内部组件。



Cloudera Manager

第一个想到的HBase管理工具就是Cloudera Manager，简称CM（是的，可能我们有点主观）。Cloudera Manager是可用的、功能最完备的管理工具。Cloudera Manager具有先入优势，在开发方面相对于Apache Ambari有两年的领先期。CM最主要的缺点就是闭源。但好消息是，CM有许多神奇的功能包括单击安装界面，使得安装Hadoop集群很简单。CM最有用的功能是parcel、tsquery语言和分布式日志搜索。

Parcel是安装Cloudera的Apache Hadoop（CDH）发行版的选项。极简形式下的Parcel就是美化了的压缩包。Tarball就是Cloudera打包了所有必要的start/stop脚本、libs /jar和CM功能正常运行所必需的文件。Cloudera Manager可使用此设置来轻松管理已部署的服务。图3-1展示了Cloudera Manager中可用的以及已安装的服务列表清单。Parcel允许全栈滚动升级和重新启动，小版本之间的升级也不需要停机。而且它们也包含了必要的依赖，比使用packags安装Hadoop更清洁。Cloudera Manager利用统一的目录结构来生成和简化classpath以及配置管理不同的项目。

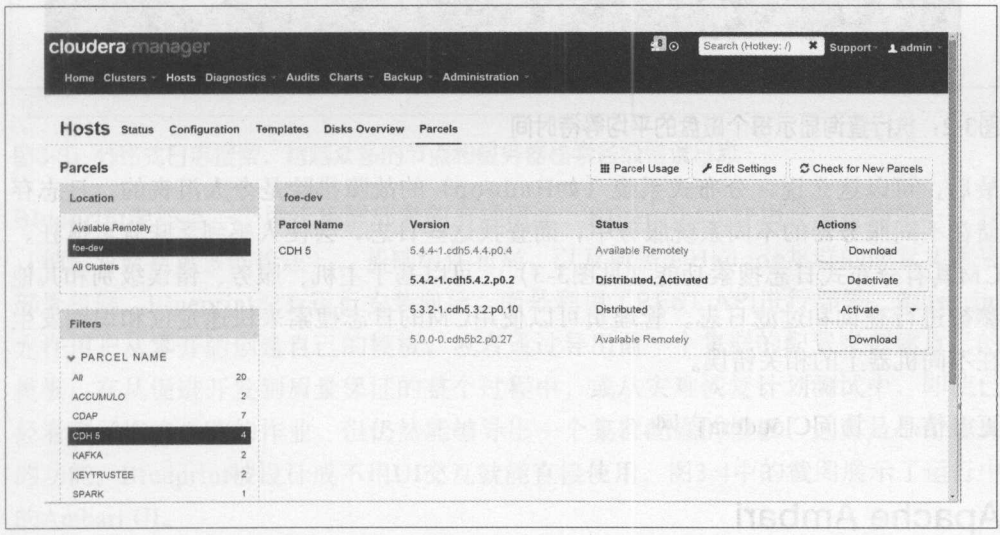


图3-1：目前可用及已部署的package列表

对于任何成功部署的Hadoop和HBase集群，监控是关键。CM内置了令人印象深刻的监控和图表功能，它还有一个被称之为“Tsquery” 的类SQL语言。如图3-2所示，管理员可以使用Tsquery建立并分享自定义图表、曲线图和仪表板来监控和分析集群的性能指标。

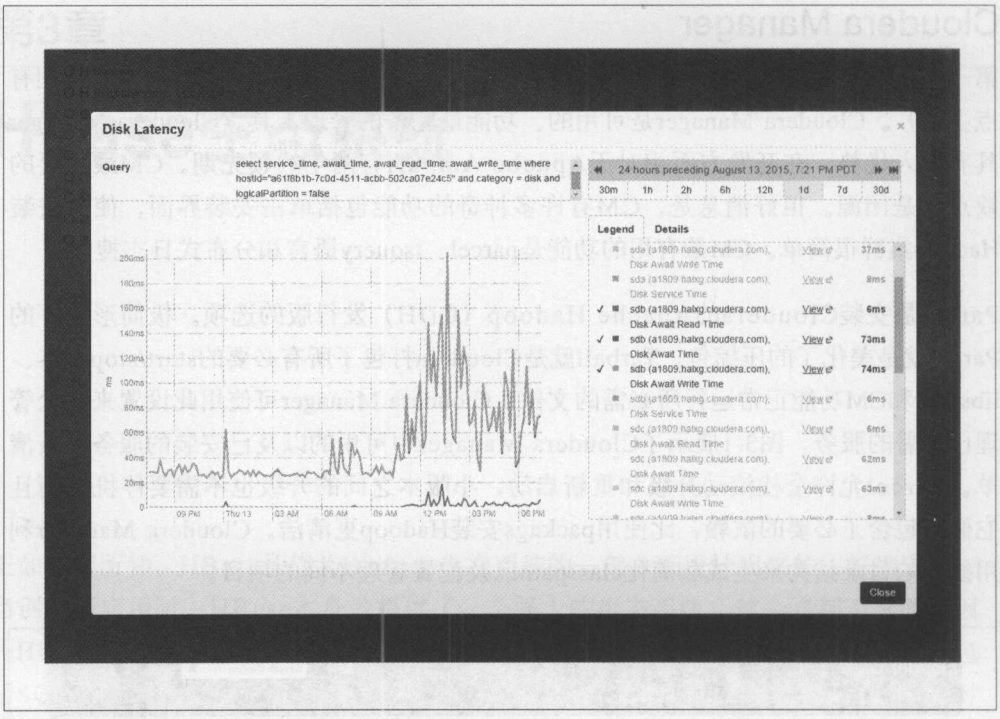


图3-2：执行查询显示每个磁盘的平均等待时间

最后，可以这么说，分布式系统（如Hadoop）的故障排除是令人沮丧的。日志存储在不同服务器的不同系统服务中，而查找这些日志，会使人感到乏味直至抓狂。CM具有分布式日志搜索功能（见图3-3），可以基于主机、服务、错误级别和其他参数进行查询和过滤日志。管理员可以使用CM的日志搜索来快速定位和识别发生在不同机器上的相关错误。

更多信息，访问Cloudera官网。

Apache Ambari

Apache Ambari是一个类似开源的Cloudera Manager。Ambari是Apache基金会的一部分，在2013年12月成为顶级项目。Ambari配备了所有你所期待的管理工具，它能够部署、管理和监控Hadoop集群。Ambari有一些很有趣的特性，帮助其脱颖而出成为一流的管理工具。它最显著特征是被称为Blueprint的部署模板，有着类似Stack的可扩展框架和Tez那样的用户视图。

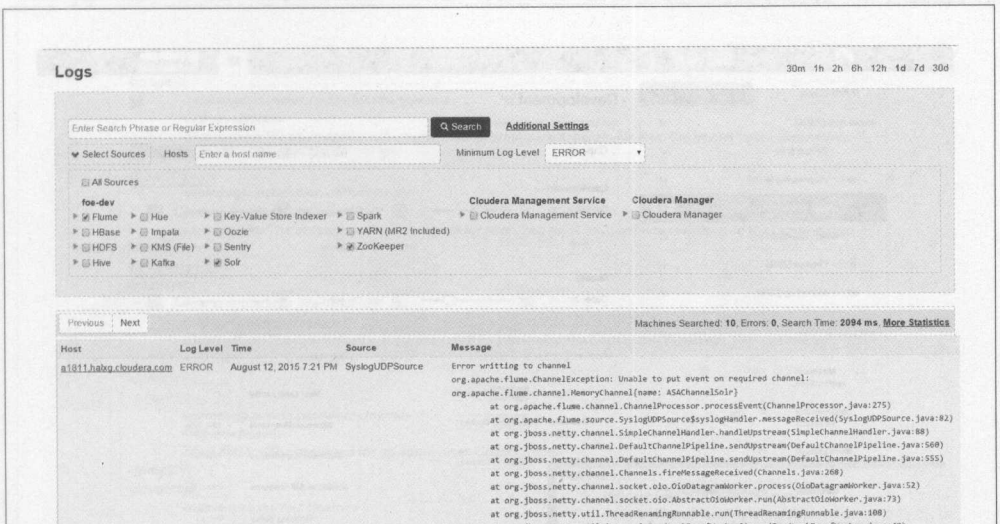


图3-3：分布式日志搜索，跨越众多的节点和服务器检索各级错误日志

Blueprint架构是一个用于集群部署的有趣概念。Blueprint允许用户通过指定一个栈（稍后对此有更多的论述）、基础组件布局，以及所需的Hadoop集群的配置实例来部署集群。Blueprint没有运行安装向导，而是利用了REST API进行部署。Blueprint允许用户从零开始创建自己的模板，或者通过导出前一个集群的配置来创建自己的模板。在从促进开发到质量保证的整个过程中，或从灾难恢复计划测试中，即使已经看到了许多失败的作业，但仍然能够导出一个集群配置的模板，这真是一个惊人的功能。Blueprint被设计成不用UI交互就能直接使用，图3-4中的截图展示了运行中的Ambari UI。

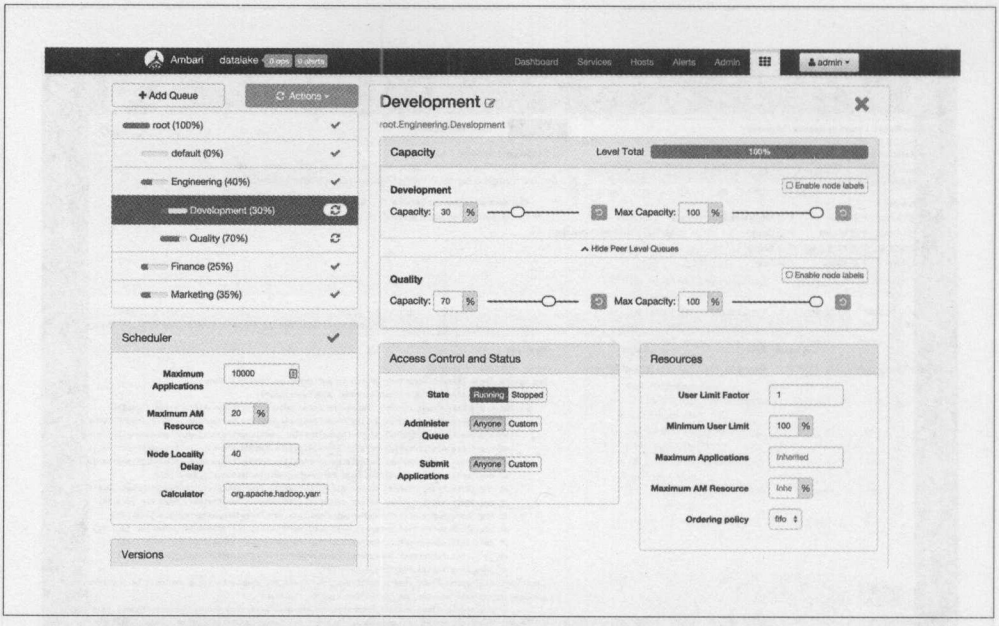


图3-4：简便的含滚动条并可切换的YARN调度部署

Ambari提供了一个称为Stack的可扩展框架。Stack允许用户快速扩展Ambari来使用定制服务。Stack允许额外的定制服务和客户端服务，并可将其添加到Ambari服务器以便于管理。

其中，最惊人的功能是，通过整合管理源码使得正在运行的独立于核心HDP（Hortonworks数据平台）的服务更易发布。这里可以举两个例子，比如Pivotal HAWQ或Apache Cassandra。我们在ZData公司的同仁已经做了大量的工作，并且把Greenplum添加到Ambari中作为一个例子发送了过来。

如图3-5所示，从左上方顺时针看：第一个图为创建一个配置，第二个图为创建服务，最后的图为创建start/stop脚本。

```
<configuration>
        <property>
                <name>gp.installer.zip.file.location</name>
                <value></value>
                <description>The absolute file path of where the Greenplum installer zip file is on the master host.</description>
        </property>
        <property>
                <name>gp.installation.path</name>
                <value>/usr/local</value>
                <description>The absolute path to the install location. You must have write permissions to the location you
specify.</description>
        </property>
        <property>
                <name>gp.admin.user</name>
                <value>gpadmin</value>
                <description>The Greenplum system user used to administer the Greenplum Database. The user will be
created on all Greenplum hosts.</description>
        </property>
        <property>
                <name>gp.admin.password</name>
                <value></value>
                <description>The password for gp.admin.user.</description>
        </property>
        <property>
                <name>use.mirrors</name>
                <value>false</value>
                <description>Create segment mirrors</description>
        </property>
</configuration>
```

```
import sys
from resource_management import *
class Slave(Script):
  def install(self, env):
    print 'Install the Sample Srv Slave';
  def stop(self, env):
    print 'Stop the Sample Srv Slave';
  def start(self, env):
    print 'Start the Sample Srv Slave';
  def status(self, env):
    print 'Status of the Sample Srv Slave';
  def configure(self, env):
    print 'Configure the Sample Srv Slave';
if __name__ == "__main__":
  Slave().execute()
```

```
<service>
        <name>GREENPLUM</name>
        <displayName>Greenplum</displayName>
        <comment>Pivotal Greenplum Database</comment>
        <version>0.1</version>

        <components>
          <component>
            <name>GREENPLUM_MASTER</name>
            <displayName>Greenplum Master</displayName>
            <category>MASTER</category>
            <cardinality>1</cardinality>
            <commandScript>
              <script>scripts/master.py</script>
              <scriptType>PYTHON</scriptType>
              <timeout>4800</timeout>
            </commandScript>
          </component>

          <component>
            <name>GREENPLUM_SLAVE</name>
            <displayName>Greenplum Segment</displayName>
            <category>SLAVE</category>
            <cardinality>1+</cardinality>
            <commandScript>
              <script>scripts/segment.py</script>
              <scriptType>PYTHON</scriptType>
              <timeout>600</timeout>
            </commandScript>
```

图3-5：添加Greenplum配置示例

在图3-6中，屏幕顶部是已经安装的服务和Ambari服务器接受到的心跳信息。底部的截图是选择自定义Stack来安装，该自定义Stack可通过自定义配置获得。

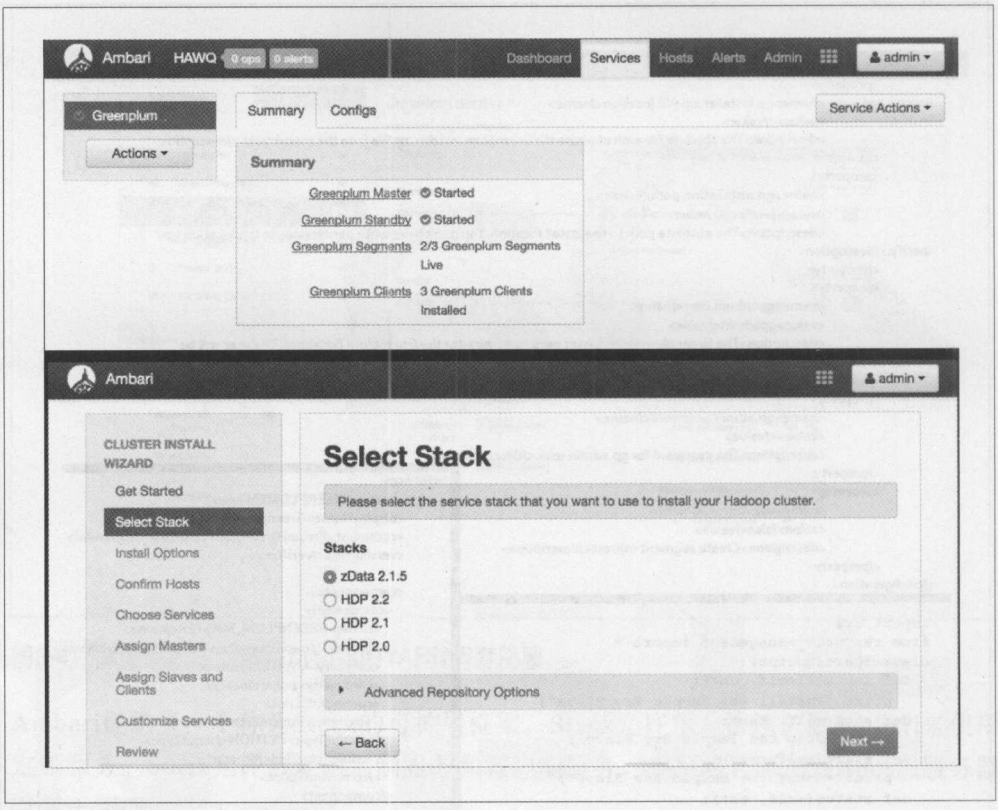


图3-6：将自定义stack加载到UI

通过User Views来使数据可视化是一件令人激动的事情。和Stack一样，Ambari Views能够集成第三方UI源码插件来管理数据以及Ambari stock没有的服务。此外，Ambari非常适合在用User Views中捆绑几个即插即用的DAG流程工具，让你开始使用。Ambari配备了Capacity Scheduler View和Tez view。在Ambari所有的功能中，Tez View是迄今为止最酷的功能，它会显示哪些可能会下线（见图3-7）。Tez View可提供正在运行的任务的DAG可视化视图。已运行的每一个作业都可以进入到一个可视化层，快速显示Job中每部分运行了多少个任务，便于快速实现对数据倾斜问题的优化。在这项功能可用之前，要优化这些作业，需要对许多的作业和日志进行深入的挖掘。

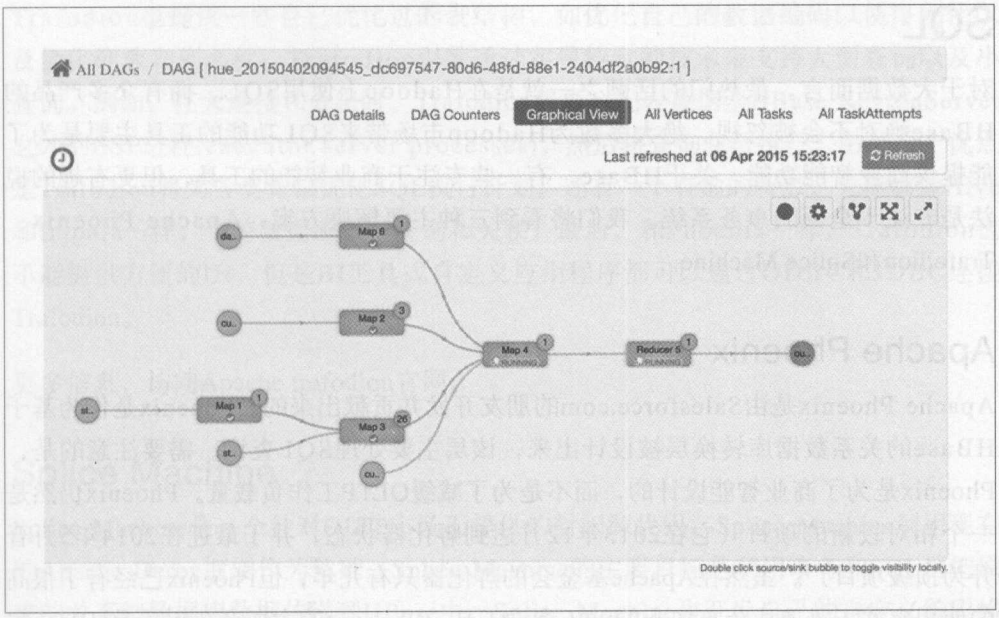


图3-7：查看Tez 任务实际执行图

更多信息，请访问Apache Ambari官网。

Hannibal

我们想提及的最后一个管理工具不是一个独立的工具，而是对各种环境的补充，无论是CM、Ambari或一个独立的Puppet/Chef基础设施。Hannibal是一个从不同的方面展示HBase内部架构的工具，如region的分布，每个表中region的大小以及随时间推移region的演变。这是非常强大的可视化信息。Hannibal展示的所有信息都能够在HBase的日志中获得，手动很难解析出来。Hannibal在下面几个方面大展身手，如手动分割region，因为它允许用户快速可视化表中最大的region，并确定什么时候是分割region的最好时间。region的历史变化也是一个非常迷人的特征，因为它可以让你查看过去一段时间内的region行为（例如HFile计数、大小和Memstore大小）。当添加额外的负载或节点到集群来验证获取的预期行为时，这是非常有用的。现在的坏消息是：似乎Hannibal已经不再被开发。然而，因为Hannibal仍在HBase 0.98以及更早版本中使用，所以我们觉得还是值得在此列入。

更多信息请访问Hannibal GitHub官网。

SQL

对于大数据而言，最热门的话题之一就是在Hadoop上使用SQL。拥有众多产品的HBase绝对不会被忽视。绝大多数为Hadoop市场带来SQL功能的工具主要是为了能提供商业智能功能。关于HBase，有一些专注于商业智能的工具，但更有趣的说法是专注于全面的事务系统。我们将看到三种主要解决方案：Apache Phoenix、Trafodion和Splice Machine。

Apache Phoenix

Apache Phoenix是由Salesforce.com的朋友开发并贡献出来的。Phoenix是作为基于HBase的关系数据库转换层被设计出来，该层主要处理SQL查询。需要注意的是，Phoenix是为了商业智能设计的，而不是为了减缓OLTP工作负载量。Phoenix仍然是一个相对较新的项目（它在2013年12月达到孵化器状态，并于最近在2014年5月晋升为顶级项目）。虽然在Apache基金会的孵化器只有几年，但Phoenix已经有了很高的使用率并正迅速成为事实上的SQL查询工具。Phoenix的主要竞争对手是HBase主要SQL查询工具Hive和Impala。Phoenix能够被构建为一个优秀工具的原因在于其更紧密地集成利用HBase协处理器、范围扫描和自定义过滤器。Hive和Impala都是创建出来用来扫描HDFS上所有的文件，这会大大影响性能，因为HBase是专为单点查询和范围扫描设计的。

Phoenix提供了一些有趣的便捷功能。二级索引可能是我们在HBase中收到的最普遍的索引请求，Phoenix提供了三种不同类型的索引：功能索引、全局索引和本地索引。每个索引类型针对不同类型的工作负载；无论它是读频繁、写频繁，还是需要通过表达式访问，Phoenix都有一个索引供你使用。Phoenix也可以处理表结构，这对于时间序列记录特别有用。通过对key增加salted bucket，让Phoenix管理这些key，能够让你避免遇到麻烦的热点问题，这些热点问题在处理时间序列数据时会带来很多麻烦。

更多信息请访问Apache Phoenix官网。

Apache Trafodion

Trafodion是一个开源工具、为HBase提供事物处理的SQL执行层，最初由惠普公司设计，目前由Apache孵化。与Phoenix不同的是，Trafodion更侧重于扩展关系模型和处理事务的过程。SQL执行层利用稳定的二级索引来提供更快的数据检索。

Trafodion也提供一些自己优化过的表结构，如优化自己的数据编码以便序列化以及优化列族和列名称。Trafodion引擎通过实现特定的技术来支持大型查询以及小查询。例如，在大连接和分组时，Trafodion将再分配表数据至HBase RegionServer之外的ESP进程(executor server processes)，然后将查询并行执行。另外一个优点是Trafodion将为各类查询优化SQL执行计划。还有一个有趣的特点，它能够像Hive和Impala一样，可跨数据源进行查询和关联。最后，和Phoenix一样，Trafodion也不能提供方便的UI，但是BI工具或自定义应用程序都可以通过ODBC和JDBC连接Trafodion。

更多信息，访问Apache trafodion官网。

Splice Machine

Splice Machine是一个针对OLTP市场而提供的封闭源代码。Splice Machine利用现有的Apache Derby框架作为基础。Splice Machine的主要目标是利用事务和SQL层把原来的关系型数据库数据传送到HBase中。Splice Machine也开发自己的自定义编码格式来对数据进行压缩，大幅压缩了存储空间并降低了检索数据的成本。和前面讨论的SQL引擎类似，并行事务处理是通过自定义的协处理器来实现的，这样保证了这些事务处理系统的高吞吐量和可扩展性。像Trafodion一样，Splice Machine也可以利用SQL执行层从HBase、Hive和Impala数据中生成商业报告。

更多信息请访问Splice Machine官网。

值得一提 (Kylin, Themis, Tephra, Hive和Impala)

还有许多其他的系统能给HBase带来SQL和事务功能：

- ApacheKylin（最初由eBayg贡献）旨在用于多维在线分析处理（MOLAP）和关系在线分析处理（ROLAP）。有趣的是因为Kylin的多维数据集必须预先创建在hive中，然后推到HBase。Kylin专为大规模报告而设计，而不是实时摄取/服务系统。
- THEMIS是基于HBase的跨行、跨表事务处理系统，由XiaoMi发起并赞助。
- Tephra是另一个事务处理系统，由Cask团队给我们带来。
- 最后，Hive和Apache的Impale是存储引擎，它们被设计用来对HDFS上的数据进行全表扫描和分区扫描。Hive和Impale都有HBase的存储处理器，程序允许

它们连接HBase和执行SQL查询。与其他系统相比，它们往往会扫描更多的数据，这将大大增加查询时间。当只涉及HBase中的一小部分或查询不受SLA限制时，Hive或impala的存在才有意义。

框架

HBase生态系统的另一个有趣的方面是框架。市场上有很多关于HBase的不同类型的可用框架。一些框架专门处理时间序列数据，而其他部分框架关注于最佳实践的编码，不同语言的使用处理以及异步客户端模型的切换。在开始讨论Kite、HappyBase和AsyncHBase之前，我们将深入了解OpenTSDB框架，该框架实际上是一个独立的应用程序。

OpenTSDB

OpenTSDB是我们将要讨论的最有趣的应用之一。OpenTSDB本身不是一个框架，而是代替了处理时间序列数据的自定义框架。OpenTSDB被设计为基于HBase之上的一个时间序列数据库。它是集数据查询和数据呈现于一体的完整应用层，同时也可以使用它集成时间序列数据，这是一个非常简单的系统，独立的、无状态收集器被叫做时间序列守护进程（TSD），用来从服务器接收时间序列数据。然后，那些TSD作为客户端，在一段时间内通过提取串行数据的模式将数据以一种有效方式写入HBase。终端用户绝对不允许访问任何的HBase数据。所有必要的调用都需要通过TSD。OpenTSDB还内置了一个UI（见图3-8），用于从时间序列数据中绘制不同的指标。TSD还可以公开一个HTTP API，以方便第三方监视框架或报告仪表板查询数据。

更多信息请访问OpenTSDB官网。

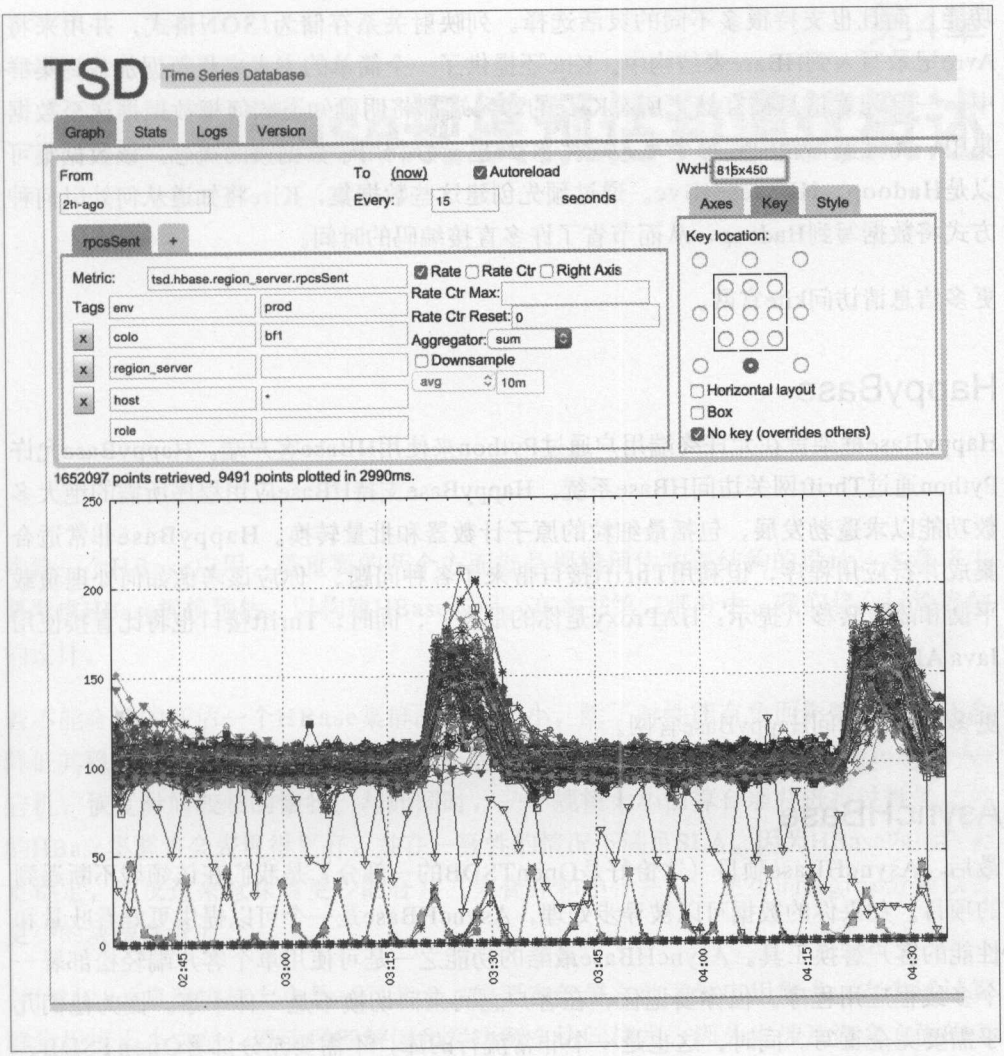


图3-8：OpenTSDB提供一个全功能的方便的UI

Kite

目前有很多不同的框架可以替代或改变HBase客户端行为。Kite是一个用于编写Hadoop最佳实践的完整SDK。Kite还带有一个被称为morphlines的预先构建库，该库涵盖预转化、汇总、抽取等操作。HBase Kite相对较新，而且仍然处于实验状态。在Cloudera的客户操作工具中大量使用了Kite中设计用于HBase的那部分功能。该功能基于我们众所周知的HBase-common内部库。其目的是如何编写在HBase列中存储记录的代码，以优化存储空间以及提高执行效率。Kite提供与HBase列映射的

功能，而且也支持很多不同的灵活选择。列映射关系存储为JSON格式，并用来将Avro记录写入到HBase表结构中。Kite还提供了一个简单的方式来将数据加载到集群中。一旦配置信息被存储，所有Kite的写入端都将明确知道如何把数据推送至数据集中，而无需重新配置每个写入端。 Kite也充分利用了数据集的概念。该数据集可以是Hadoop、HBase或Hive。通过预先创建这些数据集，Kite将知道从何处以何种方式将数据写到Hadoop，从而节省了许多直接编码的时间。

更多信息请访问kite官网。

HappyBase

HappyBase框架旨在允许终端用户通过Python来使用HBase客户端。HappyBase允许Python通过Thrift网关访问HBase系统。HappyBase支持HBase应用程序所需的绝大多数功能以求蓬勃发展，包括最细粒的原子计数器和批量转换。HappyBase非常适合集成并行应用程序，但利用Thrift接口带来了各种问题。 你应该考虑如何处理负载平衡和故障转移（提示：HAProxy是你的朋友），同时，Thrift接口也将比直接使用Java API慢。

更多信息请访问HappyBase官网。

AsyncHBase

最后，AsyncHBase项目（这恰好是OpenTSDB的一部分）是我们在该领域不断遇到的项目。如果你的数据可以被异步处理，AsyncHBase是一个可以提供更好吞吐量和性能的客户替换工具。AsyncHBase最酷的功能之一是可使用单个客户端轻松部署一个多线程应用程序。但你要记住，在客户之间来回切换不是一件小事，因为他们几乎需要完全重写。同时，这也是一个非常流行的库，不需要充分部署OpenTSDB，就能处理时间序列数据。

更多信息请访问OpenTSDB的GitHub。

第4章

HBase规模预估和调优概述

构建一个HBase应用，最重要的两个方面就是规模预估和表结构的设计。本章将主要考虑HBase规模预估，以构建HBase应用。在本书第二部分中，我们将会讨论表结构设计。

若不能合理的评估一个HBase集群的规模大小，除了对性能有负面影响之外，还会降低其稳定性。很多集群由于不够大开始慢慢出现客户端响应超时，RegionServer宕机，恢复时间较长的惨状。与此同时，一个规模大小估算合理并进行过性能调优的HBase集群将会表现得更好，并在一致性的情况下满足SLA，因为HBase内部将会更稳定，这反过来意味着更少的合并（大合并和小合并）、更少的region分裂以及更少的块缓存流失。

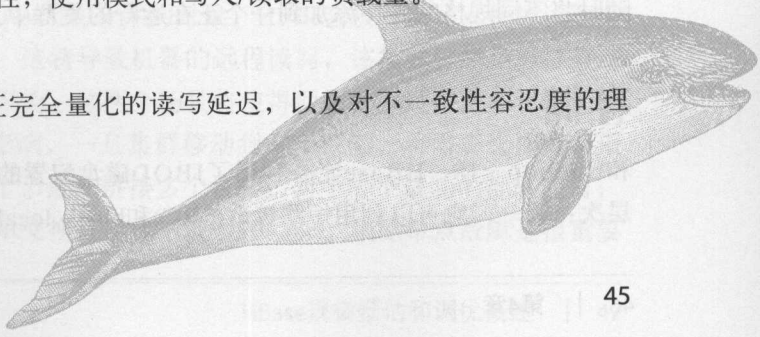
评估HBase集群规模大小是一门艺术，需要在部署之前了解应用需求。在尝试计算群集规模大小之前，要确保理解读和写这两个访问模式。因为它涉及许多方面的考虑，合理地评估HBase大小是一件具有挑战的事情。在开始评估集群规模之前，分析项目的要求是很重要的。这些要求应分为三类：

负载

这就需要了解一般的并发性，使用模式和写入/读取的负载量。

服务水平协议（SLA）

你应该有一个SLA来保证完全量化的读写延迟，以及对不一致性容忍度的理解。



容量

你需要考虑每天摄取多少数据，项目生命期中总数据大小以及总的数据持续时间。

充分了解该项目的这些要求后，接下来我们可以计算集群的大小。由于HBase依赖于HDFS，采取自下而上的方式设计一个HBase集群是很重要的。这意味着从硬件开始，然后网络、操作系统、HDFS，最后HBase。

硬件

和大的、多租户部署相比，只部署HBase所需要的硬件成本开销是很少的。目前，HBase可以在内存中使用约16~24 GB的heap，尽管随着Java 7以及G1GC算法的改进将会带来一些改变。G1GC收集器的初始测试已经显示保证超过100GB的heap的较好结果是可以计算的。这在尝试垂直扩展HBase时是特别重要的，具体的计算方式我们在后面讨论。

当使用超过24GB的heap空间时，垃圾回收器将会暂停并将持续很长（30秒或更多），以至于RegionServer因为zookeeper引发的错误而导致失败。目前，为DataNode节点配置64GB到128GB的内存足够满足RegionServer、DataNode和操作系统的要求，同时留下足够的空间让块缓存辅助数据读取。在偏读取的环境中，依靠利用BucketCache，256GB或者更多的内存将会更有优势。从CPU的角度来看，HBase集群的总核数不需要很多，所以中等主频的低端核数是相当足够了。截至2016年年初，低端市场上标准商品机的配置通常为双octacore，或者是处理器主频大约是2.5GHz的双dodecacore。如果你正在创建一个多租户集群（尤其是如果你打算优化MapReduce），使用更多的核数是很有益处的（具体数量取决于你的使用情况和使用的组件）。多租户的用例是非常难处理的，尤其是当HBase受制于高要求的SLA，测试MapReduce工作以及HBase负荷都是很重要的。为了保证SLA，有时需要有一个批处理集群和实时集群。但随着技术的成熟，这也将变得越来越没有必要。Hadoop、HBase集群的最大优势之一在于不需要一个同质的环境（尽管你的理智推荐这样做）。这使得在购买硬件时有很大的灵活性，并允许你在需要的时候可随时将不同规格的硬件添加到一个正在运行的集群中。

存储

和Hadoop一样，HBase充分利用了JBOD磁盘配置的优势，其主要优点体现在两个层次：第一，它可以利用短路读取(SRC)和block locality以获取更好的性能；第二，

它有助于通过减少昂贵的RAID控制器的数量以降低硬件成本。磁盘数量目前不是一个纯HBase集群的主要因素（没有MR，没有Impala，没有Solr，或任何其他正在运行的应用）。HBase每个节点配置8～12块磁盘，就可表现良好的功能。和可用性相比，HBase偏向于数据的一致性，因此HBase对写入路径做了一些限制。HBase已经实现了提前写日志（WAL），当数据写入的时候必须先写入日志。因为WAL记录了每一次写入磁盘的数据信息，因此，通过API，REST和Thrift接口写入后处理速度比较慢，导致在一个驱动器上造成写入的瓶颈。规划将来的磁盘数量对于在HBASE-10278所做的工作来说将变得越来越重要。对增加多WAL支持来说，这是一个上游的JIRA，它将数据的写入量分散到每个节点的磁盘上消除WAL写瓶颈。HBase和Hadoop有相似的思路，其中SSD目前已经过度使用，并且从部署的角度看并没有必要。当你面向一个永不过时的长期的HBase集群时，采购25%~50%的存储将会是有益的，这可以完全应对系统的多WAL和档案存储。

档案存储

有关档案存储的更多信息，请参见Apache软件基金会的有关档案存储、SSD和内存的指南。

当处理要求高的SLA的时候，我们建议只有增加SSD硬盘，尽管它们每GB的价格是相当昂贵的。

网络

在设计一个HBase集群的时候，网络是一个重要考虑因素。HBase集群通常使用标准的千兆位以太网（1 GbE）和10千兆以太网（10 GbE），而不是更昂贵的替代品（如光纤或无限宽带）。尽管1 GbE绑定是最小的推荐配置，考虑到未来扩展，10 GbE则是理想配置。如果用于绑定两个或四个端口，那推荐的最小配置是1 GbE。由于硬件和软件两方面都会持续按照比例扩展，所以10 GbE是理想的配置（目前来看，将来配置10GB也比较容易）。而且在系统硬件发生故障及做大合并的时候，10 GbE的配置表现突出。在硬件故障期间，底层的HDFS必须重新复制该节点上存储的所有数据。同时，系统在做Region大的合并的时候，HFile会被重新写入托管该数据的RegionServer的本地磁盘中。这将导致机器的远程读写，该集群已经经历过节点故障或者节点上Region的重新平衡。这两个场景将使得网络处于过度饱和状态，它可能对高要求的SLA造成性能影响。一旦集群移动到多个机架，将需要选择机架顶部（TOR）的交换机来连接多个节点并桥接多个机架。对于一个成功的部署而言，内部机架连接（见图4-1）的TOR交换机速度不低于10 GbE。消除单点故障是很重要

的，因此强烈推荐机架之间做冗余。这样即使Hadoop和HBase中的一个机架故障，集群也能够正常工作，但它不会是一个有趣的经历。如果群集规模变得太大，以至于HBase需要在数据中心中跨过多个通道，这样的话，核心/汇聚交换机可能需要被引入。这些交换机的运行速度不能低于40 GbE，同时也建议做数据冗余。集群应在自己的网络和网络设备上进行隔离。Hadoop和HBase可以迅速占满整个网络，所以在其自身的网络中分离集群可以帮助确保HBase不会影响数据中心的任何其他系统。为管理方便和安全，在集群的网络上也可能实现VLAN。

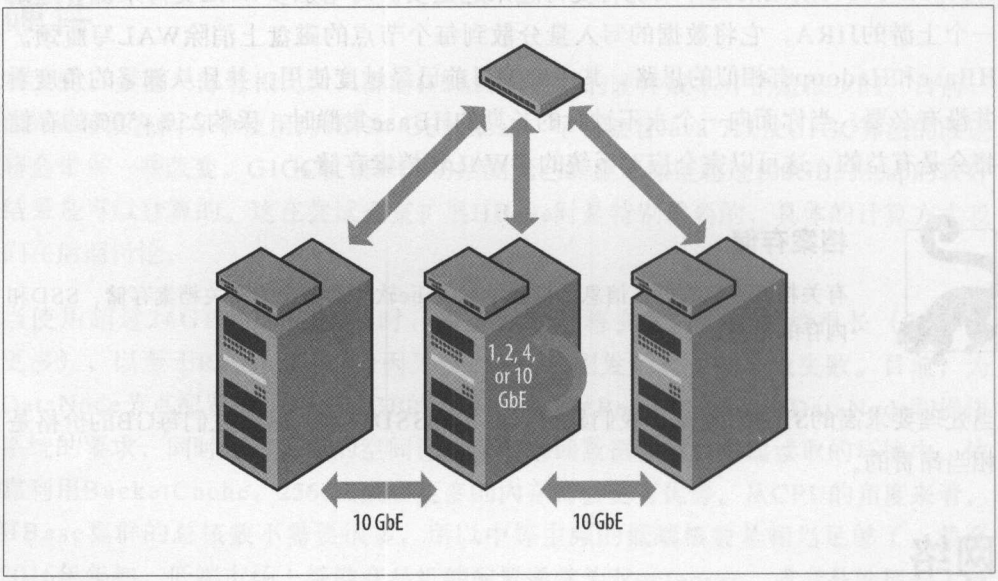


图4-1：网络示例

操作系统调优

Hadoop、HBase的操作系统没有很多特殊的考虑因素。Hadoop、HBase集群标准的操作系统是基于Linux系统的。对于任何生产的部署来说，使用的是企业级的发行版（例如，RHEL、CentOS、Debian、Ubuntu或SUSE）。Hadoop将其块数据直接写入操作系统的文件系统。任何新的操作系统版本，建议使用EXT4格式的本地文件系统。XFS文件系统是可以接受的，但它还没有被广泛部署在生产环境中。对操作系统来说，需要考虑的下一组因素是交换和交换空间。对于HBase来说，不使用进程内核来进行交换，显得非常重要。内核和进程在HBase节点的交换可能会引起严重的性能问题，从而导致故障。建议通过设置分区大小为0，禁用交换空间。通过将vm.swappiness 设置成0或1，禁止进程交换。

Hadoop调优

只有比Apache HBase 0.98更新的版本才会支持Hadoop2的YARN。意识到YARN(MR2)对HBase集群的影响是很重要的。MapReduce的工作负荷造成HBase资源短缺的无数例子已经被Cloudera支持并处理了。由于资源等待被释放，将导致长时间的服务停顿。

由于资源管理器被移动到YARN的架构中，带有YARN 的Hadoop2.0变得额外复杂。YARN还支持托管服务之间资源的更多细粒度级别的调整。上游有一些工作要将HBase整合到YARN框架中（HBase on YARN，或HOYA），但它目前没完全做完。这意味着必须通过YARN调优来满足HBase运行的稳定，避免资源短缺的问题。YARN允许对CPU利用和内存消耗做特定的调优。三个主要功能需要考虑：

`yarn.nodemanager.resource.cpu-vcores`
: container允许的CPU核个数。

`yarn.nodemanager.resource.memory-mb`
: 分配给container使用的，物理内存的大小，以MB为单位。使用HBase的时候，重要的是不要为节点过度分配内存。建议分配8~16GB给操作系统，2~4GB给DataNode，12~24GB给HBase，其他部分给YARN（假设没有其他工作负载，如Impala或Solr）。

`yarn.scheduler.minimum-allocation-mb`
: 每一个容器的请求在RM的最低配置，以MB为单位。内存请求低于此配置将不会生效，指定的值将被设置成最小的分配单元。推荐的值大小介于1~2GB。

当执行计算时，重要的是要记住，绝大多数的服务器使用英特尔芯片组的超线程，超线程允许操作系统为每一个物理核心配置两个虚拟的或逻辑的核心。下面是一些在HBase中使用YARN组件的调优贴士（这将取决于工作负载的大小，你的实际参数可能有所不同）：

> physicalCore × 1.5 = v-core总数

我们还必须记住给HBase预留离留些空间：

> v-core总数 – 1 （HBase） – 1 （DataNode） – 1 （NodeManager） – 1 （OS） ...– 1（其他的服务，例如Impala或者Solr）

对于一个拥有双10核处理器的HBase与YARN集群，调优如下：

20 × 1.5 = 30 virtual CPU
30-1-1-1-1 = 26 v-core

在使用HBase与YARN的集群中，一些消息来源建议要进一步将26个核对半分。你应该在生产环境中测试验证你应用的最佳调优参数。

HBase调优

正如本章开头部分，在评估HBase的规模时，需要对项目三个方面的需求深入理解：任务负载量、SLA和容量。任务负载量和SLA可以分为三大类：读、写或混合。从硬件的采购角度来看，每一类都有自己的一套可调参数、SLA要求以及规模评估方面的顾虑。

第一个需要审视的是写任务繁重的负载。数据导入HBase的方式主要有两种：通过一个API（Java，Thrift或REST）或通过使用批量加载。这是一个重要的区分，如API将使用WAL和Memstore，而批量加载是短路写并绕过上述两个。正如在本章前面提到的“硬件”，HBase的主要瓶颈是Memstore带来的WAL。在API驱动的写入模式下，有一些快速的公式可以用来确定最佳的写入性能。

确定每个节点的region数：

HBaseHeap × memstoreUpperLimit = 可获得的MemstoreHeap
可获得的MemstoreHeap/ memstoreSize = 推荐活动的region个数

（这实际上是基于列族的，该公式基于一个列族）

确定每个节点的原始空间：

推荐的region个数 × maxfileSize × replicationFactor= 原始使用空间

要确定保持WAL数目：

可获得的MemstoreHeap /（wal大小 × wal乘数）= wal个数

下面是一个使用现实世界的数字计算规模大小的例子：

HBase heap = 16 GB
Memstore upper limit = 0.5*
Memstore size = 128 MB

Maximum file size = 20 GB

WAL size = 128 MB

WAL rolling multiplier = 0.95

首先，我们将确定每个节点的region个数：

16384MB × 0.5 = 8192

8192 MB / 128 MB = 64 activeRegion

下一步，我们将确定每个节点的原始空间：

64 activeRegion × 20GB × 3 = 3.75 TB

最后，我们需要确定保留项WAL的数量：

8192 GB / (128 MB × 0.95) = 67 WAL

使用这个公式时，建议每个RegionServer不超过67个region，同时保持WAL总数不超过67个。这将在每个节点使用3.7 TB（包括副本）来存储HFile文件，其中不包括用于快照或合并的空间。

批量加载是一种并不通过WAL和memstore的短路写入方式。当使用completebulkload工具的时候，在MapReduce job的rudece阶段，HFile被completebulkload工具创建和加载。region数量限制大部分来自WAL和Memstore。当使用bulkload的时候，region的个数受制于索引的大小及响应的时间。建议把测试读取响应时间尺度扩大到100个 region；之前从未听说过一个RegionServer能够成功部署150~200个region。使用批量加载的时候，降低上限（HBase.regionserver.global.memstore.upper限制）和下限（HBase.regionserver.global.memstore.lower限制）分别下降到0.11和0.10是很重要的，然后根据可用的heap将block cache提高到0.6~0.7。对memsore和block cache的调优将允许HBase在内存中保存更多的数据以便数据读取。

需要注意的是，这些值是基于Memstore和block cache的heap百分比。

一旦所需的region数通过适当的测试已经确定，对于每个节点计算来说，就可以使用从前一样的使用空间。

同样，确定每个节点原始空间的计算公式如下：

建议region个数 × 文件最大值 × 复制因子= 原始使用空间的最大值

使用这个公式，我们会在这个实例中计算每个节点的原始空间：

150 个活动region × 20GB × 3 = 9 TB使用空间

容量是最容易计算的部分。一旦任务负载量已经明确并得到正确的验证（注意到曾经的一个测试主题了吗），它就变成了一个简单的问题，通过每个节点容量来划分数据总量即可。重要的是要预留一定的可用空间（一般建议是20%~30%的开销）。这个可用空间将用于MapReduce计算的暂存空间，以及额外的快照和大合并。此外，当region总数较高时，如果使用快照，将务必通过相应的测试以确保有足够的空间来满足集群的备份机制，因为快照可以迅速占用空间。

负载不均调优

对于以数据读取为主的工作任务，理解并测试SLA的要求是很重要的。需要严格的测试，才能确定最佳的设置。需要调整的主要配置与写负载相同（即降低Memstore中设置和提高块缓存，允许更多的数据被存储在内存中）。在HBase 0.96中引入了BucketCache。它不仅允许数据被存储在内存而且可以存储在硬盘（SSD /低延迟闪存卡）。因为HBASE-11678，你应该使用0.98.6或更高版本。

在混合工作负载环境下，HBase BucketCache变得更加有趣，因为它允许HBase在保持一个较高的读取响应时间的同时，还允许系统进行多路写入。在混合环境运行时，构建一个包含冗余资源的集群是很有必要的，因为这样做能够允许一个或两个RegionServer损失。建议在估算集群大小的时候，让每一个机架多出一个额外的RegionServer。这也将给集群提供更多的内存空间用于数据的读取和额外的WAL写入，以获得更好的响应时间。再次，若需从适当的任务负载中找到完美的平衡，也就意味着我们需要对WAL数量、memstore限制，以及分配的块缓存总数进行适当调整。

HBase在集成使用其他Hadoop组件时，可能会遇到诸多问题。我们将在第二部分部署HBase与Impala、Solr和Spark。不幸的是，真的没有一个很好的解决方案能够在HBase中同时使用多个组件。出现的一般问题包括以下几个方面：

- CPU 竞争。
- Memory 冲突。

- I/O 竞争。

通过对YARN做适当的调优（Spark运行在YARN上）以及设置相应的资源冗余，前面两个是相当容易处理的。我的意思是如果你知道你的应用程序需要将W GB配置给YARN，Impala将使用X GB，HBase heap需要Y GB 来满足SLA需求，我想保留ZRAM给操作系统，那么我的公式应该是：

W + X + Y + Z = 总共需要的内存

这个公式乍一看是有道理的，但不幸的是，事实不会如此简单。我们建议至少提升到25%~30%的空闲内存。这种方法有两方面的好处：首先，当购买额外的节点时候，它会给你垂直增长的空间，直到这些新的节点添加到集群；其次，操作系统将利用额外的内存和操作系统缓冲区来全面的提升总体性能。

I/O竞争才真的是故事崩溃的开始。唯一可行的答案是利用Linux文件系统控制组（cgroups）。创建控制组可能很复杂，但它们可提供分配指定程序如Impala，或者最有可能的YARN/ MapReduce，到特定的I/O调节组。

大多数人不太会受到配置控制组或其他控制技术的困扰。我们倾向于建议对Impala做内存限制，对YARN做容器的限制，对Solr做内存和堆限制。然后大规模地测试你的应用程序，可以微调组件使得它们能够一起工作。

我的同事Eric Erickson，他在Solr领域比较擅长，因为每当被问到规模评估时他总是回答看情况，因此被冠以“看情况”先生。他认为，好的开始使得事情变得简单，但只有经过合理的测试才可以验证估计的大小。对于HBase和Hadoop而言，好消息是它们提供了线性的可扩展性，这样的话，测试可以在一个较小的规模进行并以此为基础横向扩展。HBase的成功部署没有秘密可言，你需要使用商业硬件，为了保证网络的简易性，可以使用以太网来代替更昂贵的技术，如光纤或无线宽带，以及保持对各个任务负载的详细了解。

第5章

环境设置

为了能够在本地环境测试并观察HBase的行为，我们将安装和配置一个独立的HBase环境用来运行实例。HBase可以运行在三种模式下，分别提供不同的行为（前两种模式主要目的用于测试，而最后一个模式将用于生产HBase集群）：

单机模式

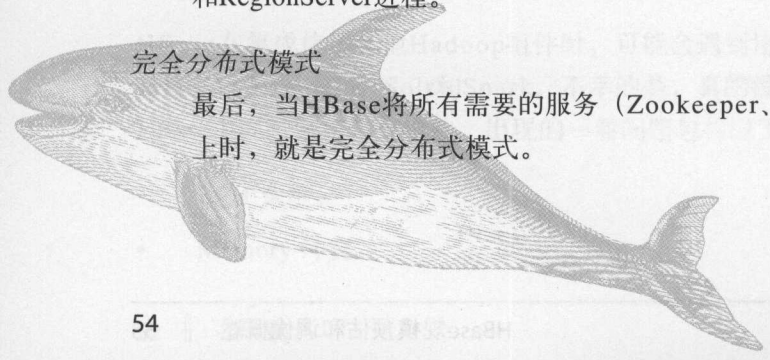
该模式将在一个虚拟机（VM）中启动所有的HBase进程。单机模式使用本地磁盘来存储HBase数据，且不需要任何特定的配置。数据将存储在当前用户配置的临时文件夹（通常是/tmp）中的一个名为HBase-$USERNAME的文件夹里面。当HBase宕机后，删除此文件夹，将会删除HBase中所有的数据。这是运行HBase最简单的模式，这也是我们将用来测试的模式。使用这种模式允许终端用户能够很容易地清除所有内容，并重启一个空白的HBase应用。这种模式也不需要运行任何其他外部服务（例如，zookeeper或HDFS）。

伪分布式模式

在伪分布式模式下，HBase可以作为一个单节点运行的集群，但是，运行和完全分布式模式下相同的进程。HBase将启动zookeeper进程、HBase Master进程和RegionServer进程。

完全分布式模式

最后，当HBase将所有需要的服务（Zookeeper、Hadoop等）运行在多台机器上时，就是完全分布式模式。



系统要求

不管你是选择一个独立的、伪分布式或完全分布式的部署，关于部署环境的选择将是其成功的关键。对于核心基础设施可以选择使用裸机或虚拟机；操作系统要安装哪个；你应该使用什么样的Hadoop发行版本；需要部署什么样的Java版本；而最重要的问题，成功部署需要什么样的系统资源。以下部分将有助于引导并回答这些问题。

操作系统

这本书提供的所有示例的输出是从装有HBase 1.0.1的Linux shell终端捕获的（一些在Debian，一些在CentOS）。为了重现我们的结果，请尝试使用与我们相同的版本。这些例子可以运行在一个物理或虚拟机上。命令也可以在其他环境中工作（Windows、Mac OS X等）或Cygwin，但我们并没有在其他环境下做测试。

如果你想安装一个Linux环境，你可以下载并安装如下的Linux发行版本：

- Debian 7+
- Ubuntu 14.04+
- RedHat 6.6+
- CentOS 6.6

这是一个不全面的清单，还有很多其他的发行版和版本号，它们中的某些应该能运行。你应该选择一个你最熟悉的版本来安装。如果你决定使用Apache Hadoop的发行版，验证并确保和Linux发行版兼容。

虚拟机

在接下来的内容中，有多种方式使用虚拟机搭建伪分布式环境来运行HBase。让我们分别看看这些选项。

虚拟机模式

本地虚拟机

可以在你的本地计算机上安装一个虚拟机支持工具（例如VirtualBox），然后创建一个Linux虚拟机，你可以在上面安装和运行HBase的单例模式或者伪分布式模式。

公有云

除了需要在公共云（如Amazon EC2）上创建Linux VM（实例）外，其他与前面的选项有点类似。然后在上面安装HBase软件，并以单机或伪分布式模式运行。你也可以提供多个虚拟节点，以完全分布式模式运行HBase。在公共云中运行HBase生产集群需要小心，因为可能需要一些特定的配置。

本地虚拟分配

在你的本地计算机上安装虚拟机支持工具（例如VirtualBox），而不用手动安装软件，你只需下载Cloudera快速启动虚拟机，它包括一个以伪分布式安装的全功能HBase（无需手动安装）。

Linux环境

如果没有一个可以用来安装Linux的计算机，可以通过虚拟机在你的本地环境中运行它。以下应用程序将帮助你在本地虚拟机中运行一个Linux环境：

- VirtualBox
- VMware
- KVM

Hadoop发行版

这本书涵盖了HBase的应用设计和开发，但要全面落实，一些实例可能需要其他外部的一些应用程序，如Solr或Flume。安装、配置和运行这些工具都超出了本书的范围。如果你想实现这些集成的例子，你将需要有一个已经安装好的全分布式Hadoop虚拟机。Hadoop发布版本是一个组件包，通常由Hadoop供应商打包，包括大多数却并不是Hadoop所有的相关应用。这样的结构通常是更易于安装，因为所有的应用程序都是被配置为一起工作的，它们也会用一些安装脚本来为安装做准备。

说它简单，是因为它已经包含了所有我们需要的应用程序（HDFS、HBase、Solr、Flume等），你可以使用Cloudera 快速启动虚拟机。我们使用VM 5.4.2版本来建立所有这些用例，并进行测试。如果你不熟悉Linux或HBase和Hadoop的安装，对你而言，使用快速启动虚拟机并使用默认的路径会容易些。

本书中所有的实例都是基于HBase 1.0.1创建的。如果你在一个虚拟的环境中选择使用Hadoop的其他发行版本，或如果你决定在本地安装HBase，需要确保你选择的发行版本和HBase版本一致。虽然这些例子可以在比较新的版本上运行，但是它们中的大多数例子可能在老版本上将无法运行。

如果你选择使用Hadoop的发行版，我们仍然推荐阅读以下部分以便熟悉HBase，包括它的约束条件以及安装方式。

为了充分使用HBase的大部分功能，本书中所有例子都分别在本地单机Linux 环境与Cloudera快速启动虚拟机环境中做过相应的测试验证比较。如果你能够访问到Hadoop集群测试环境（物理的或虚拟的），你也可以使用它。这将允许你选择你最舒适的环境来运行你的验证案例。

资源

在你选择何时启动HBase path的时候，首先需要配置一定必要的资源，足够的资源将能预先决定其成功或失败。

内存

不管在本地环境还是在虚拟机中运行，你都需要向HBase及其他服务提供足够的内存。单机模式的HBase在本地运行需要至少1GB的内存。然而，如果可能的话，我们建议分配更多的内存，因为这样做会让你在更大的数据集上运行测试，并且对应的应用程序的结果和性能将会表现的更好。如果你直接在虚拟环境下运行Hadoop分布式应用案例，你应该为虚拟机至少分配4GB的内存。再次，若给运行环境分配更多的内存，你的应用程序将会有一个更快的响应时间，并能够运行更大的数据集。这就是我们建议分配至少8GB内存空间给你的虚拟机的原因。

你应该给你的虚拟机分配尽可能多的内存。如果你的环境有32GB或者更多的内存，一定要分配尽可能多的内存给你的虚拟机。我们大部分的例子都是在12GB的VirtualBox虚拟环境上运行的。

磁盘空间

HBase的安装文件需要至少100MB（例子只需要几兆字节）的空间，所以你可以在不到1GB的可用磁盘空间运行所有的例子。然而，如果你想进一步看到例子中的所有功能，将需要创建更大的数据集，这需要更多的空间来进行测试。因此，我们建议允许至少10GB的可用的磁盘空间。

如果想在一个虚拟的环境中运行应用，你将需要至少4GB的可用磁盘空间来下载虚拟机，同时需要另外的4GB空间来做解压。Cloudera的快速启动虚拟机允许虚拟盘扩大到64GB。创建大数据集及长周运行的HBase应用，会在虚拟环境中生产很多数

据。因此，你将很容易地使用高达50GB的磁盘空间。尽管它可能开始占用的空间很少，但我们还是建议你开始之前留出空余磁盘。

Java

HBase是由Java开发的，所以它依赖于JDK的特定版本。其中HBase 0.94开始支持JDK 7，HBase 0.99 开始支持JDK 8，而从HBase 1.0开始不再支持JDK 6。运行当前HBase版本和提供的实例，我们推荐使用最新的可用版本JDK7。我们使用的虚拟机案例JDK版本为1.7.0_67 ，所以你应该使用该版本以确保获得同样的结果。

小心选择你的JVM版本，你需要确保它是没有bug的。例如，JDK 1.7.0_75在sudo模式命令下运行时JPS命令有一些问题。

Version	Recommended JDK	Other JDKs
1.x	JDK 7	JDK 8
0.98	JDK 7	JDK 6，JDK 8
0.96	JDK 7	JDK 6
0.94	JDK 6	JDK 7

HBase单机安装

这一节将描述在裸机或者在运行的Linux虚拟机上安装HBase的必要步骤。

运行单机版的HBase的应用是一个简单的过程，你只需要下载的HBase的binary安装包，解压并运行即可。

在写这本书的时候，最新的Apache HBase稳定分支版本号为1.0.1。我们建议使用镜像下载该发行版本。一旦你找到下载HBase 发行版的正确镜像路径，要下载的文件的命名应当类似*HBase-1.0.x-bin.tar.gz*，其中X是最后的subrelease号码。把下载的文件解压到目录，你可选择用`tar -zxf`命令解压。如下的命令都是在基于HBase已经被解压或者能够从`~/HBase`访问以及*HBase/bin*文件夹已被添加到用户路径的基础上进行的。

使用UNIX命令行，上述步骤可以实现如下：

```
#:~$ cd ~
```

```
#:~$ wget \
"http://www.us.apache.org/dist/HBase/HBase-1.0.1/HBase-1.0.1-bin.tar.gz"
#:~$ tar -xzf HBase-1.0.1-bin.tar.gz
#:~$ ln -s HBase-1.0.1 HBase
#:~$ rm HBase-1.0.1-bin.tar.gz
#:~$ cd HBase
#:~/HBase$ export PATH=$PATH:~/HBase/bin
```

因为这是基于HBase的1.0.1发布，如果这个版本是由一个更新的版本替代了，下载binary安装包可能会失败。去http://www.apache.org/dyn/closer.cgi/HBase/检查最新的可用版本，必要的话，更新前面的命令。

下一步是指定用于运行HBase的JDK版本（我们建议配置和开发环境中配置相同版本的JDK）。一旦你指定了JDK，你需要让HBase知道你的JDK的位置。首先，需要通过`java -version`命令确定JDK版本和位置。当你明确了JDK的位置之后，可以设置`JAVA_HOME`变量。有可能的Java位置如下面的两个例子：

```
#:~$ export JAVA_HOME=/usr # If using your Linux distribution JDK
#:~$ export JAVA_HOME=/usr/local/jdk1.7.0_60/ # If installed manually
```

当你的`JAVA_HOME`变量已经被正确定义，HBase的独立服务器可以简单地开始使用以下命令：

```
#:~$ start-HBase.sh
```

当HBase命令加载后，你也可以编辑*conf/HBase-env.sh*文件来获得这个值，而不是每次export `JAVA_HOME`变量后再启动服务器。

以上配置将允许HBase在本地环境中运行。有多种方式可以验证HBase是否正常运行。在后面的例子中，我们将看到如何使用Java代码进行检查，但为了简便起见，我们现在使用HBase shell。

运行以下命令启动HBase shell：

```
#:~$ HBase shell
```

输出应该看起来像这样：

```
HBase Shell; enter 'help<RETURN>' for list of supported commands.
Type "exit<RETURN>" to leave the HBase Shell
```

```
Version 1.1.5, r239b8045640b2e562a5568b5c744252e,\
Sun May 8 20:29:26 PDT 2016

hBase(main):001:0>
```

你可能会遇到关于属性过时或本地库丢失的一些警告信息。在这个例子中，可以忽略它们（为了简洁起见，我们将它们从输出中移除）。在一个完全分布式系统中，警告可能会对性能产生影响，但对于一个单独的HBase服务器则不需要引起关注。

在HBase shell中运行status命令，查看到HBase状态信息：

```
hBase(main):001:0> status
1 servers, 0 dead, 2.0000 average load
```

status命令的输出说明一个HBase服务器正在运行，每个RegionServer上有两个HBase region。至此，如果你的环境没有正确运行，请参考本章后面的“疑难解答”。

即使你刚刚安装了HBase，它也已经包含了两个region：一个region含HBase：meta 表（包含所有其他表的region信息）；另一个region含HBase：namespace表（其中包含可用的命名空间细节）。这两张表都是HBase系统表，因此属于HBase命名空间（关于命名空间的更多信息，请参阅第18章“HBase文件系统布局”）。

HBase单机服务器，现在可用于下一组测试。如果有特殊情况，参考本章后面的“故障排除”。

现在正在运行的环境，它可以通过HTTP的Web界面访问*http://localhost:16010*或你可以替换localhost为本地机器的主机名或IP。

HBase集群可以通过调用stop-Hbase.sh脚本来停止：

```
#:~$ stop-HBase.sh
stopping HBase.........
```

根据服务器的性能和环境的状态，正常的关闭HBase可能需要一些时间。

虚拟机中的HBase

安装并运行一个虚拟机，你首先需要在本地安装和运行一个虚拟化的环境（例如VirtualBox、KVM或虚拟机）。我们使用Debian操作系统来运行构建的实例，因为它可以作为一个Debian包获取，使用VirtualBox来完成我们的测试。下一步是下载快速入门文档，并在你拥有足够的可用空间的情况下来解压文件。然后将其导入到你的虚拟化环境中。VirtualBox的用户，这可以通过导航到“文件”菜单并选择“Import virtual application”（这可能需要一些时间来完成导入）。最后一步是配置你新导入的虚拟环境以确保有足够的内存。

当所有这些步骤都完成后，你可以开始运行你的虚拟环境了。第一次运行时，需要一些时间来执行初始化步骤，不需后续重启。当虚拟机已经启动并准备使用后，浏览器应该自动打开，一些有用的链接和信息可以在虚拟机VM主页上看见。

为了验证HBase已经正确的安装、配置、运行，只需打开一个终端，并继续进行和单机一样的安装步骤。

打开HBase shell：

```
#:~$ HBase shell
```

并使用stauts以确保它正在运行：

```
hbase(main):001:0> status
1 servers, 0 dead, 2.0000 average load
```

然后你可以关闭你刚刚打开的终端。祝贺你!你有一个伪分布式（具有集群特征但是带有单个节点）HBase，HDFS，YARN等服务运行环境。

如果出现HBase没有正在运行，但之前是正确运行的情况，它可能被停止了，因为你暂停了VM。你可以使用sudo jps命令列出所有运行Java的程序：

```
[cloudera@quickstart ~]$ sudo jps
3097 HMaster
2820 ResourceManager
1915 DataNode
2270 SecondaryNameNode
5199
4914 Bootstrap
5225
1844 QuorumPeerMain
2137 NameNode
4380 Worker
2050 JournalNode
```

```
3629 RunJar
4054 HistoryServer
6242 Jps
4151 Master
3445 RunJar
3346 ThriftServer
5171 Bootstrap
2397 Bootstrap
3213 RESTServer
4278 HRegionServer
4026 Bootstrap
2452 JobHistoryServer
2547 NodeManager
```

上面列出了你环境中正在运行的所有Java程序。在这里，我们寻找的是3097 HMaster进程和4278 HRegionServer进程。前者是HBase主服务器的进程，后者是RegionServer进程。进程名称之前的数字是进程的ID号，每次重启后都会发生变化。如果你没有看到这些进程，使用HBase-start.sh命令或者在Web界面启动HBase服务器。

本地与VM

如果你仍然还在选择使用哪种运行模式（例如本地模式、虚拟的Linux环境或快速启动VM），以及考虑每个模式的利弊。记住，无论你决定以哪种模式运行HBase，你可以随时通过再次输出这些例子来改用另一种模式运行。

本地模式

这种模式运行速度最快，并且需要分配的内存也最少。但是，如果你没有运行Linux环境，或者如果你不精通Linux shell，你可能偏向使用另一个模式。

优势

- 要求小于1 GB的磁盘空间，只有几个字节的内存。
- 快速停止和开始。

劣势

- 不允许运行需要其他工具的复杂用例。
- 需要在Linux环境下运行。

虚拟Linux环境

这种模式是本地模式和快速启动虚拟机模式之间的一个很好的折中模式。如果你没有运行一个本地Linux环境但是习惯使用命令行，并且不需要运行其他程序（Solr等）。

优势

- 允许你在非Linux系统轻松地运行HBase。
- 避免与已安装的应用程序发生冲突。

劣势

- 至少需要3 GB的内存（2 GB分配给操作系统，1GB分配给HBase）。
- 相比于本地模式启动慢。

快速启动虚拟机（或类似）

这是最消耗资源的模式。它启动缓慢，将使用更多的磁盘空间和内存。然而，需要预先配置所有的应用程序，才能完全运行本书中的实例。如果你想跳过所有的安装步骤，但仍然想有一个足够强大的运行环境，选择这个选项，这也是我们这本书用来开发实例的环境。

优势

- 让你轻松运行HBase和其他相关的应用程序（Solr，HDFS，YARN等）。
- 避免与已安装的应用程序发生冲突。

劣势

- 与其他两种模式相比，需要更多的内存（最低4GB，建议8GB以上）。
- 最慢的启动/重启模式。

故障排除

当HBase无法正确启动或出现了一些问题时，第一个需要看的地方是日志目录。在这个目录中可以找到三种类型的文件：

- 以*.out*结束的HBase脚本日志文件，包含启动过程的输出信息。

- 安全日志中的*securityauth.audit*文件，这应该是空的。
- 最后，最重要的一种我们的HBase的应用日志文件以*.log*结尾。

在单机模式下，你需要查看的是*HBase-$USERNAME-master-$HOSTNAME.log*。在伪分布式模式，你可以查看*HBase-$USERNAMEregionserver-$HOSTNAME.log*以及*HBase-$USERNAME-master-$HOSTNAME.log*。

接下来，我们讨论一些如果没有正确配置可能会导致失败的选项。

IP/Name配置

*/etc/hosts*文件允许为localhost定义IP地址。localhost条目通常分为127.0.0.1和主机的本地IP。检验*/etc/hosts*文件看起来像以下格式：

```
127.0.0.1    localhost
192.168.1.3  myhostname
```

在单机模式下，由于HBase通过远程连接启动多个进程，需要本地ssh无密码登录认证。

访问/tmp文件夹

在单机模式和伪分布式模式下，当HBase HDFS 的*root.dir*没有配置，HBase会将所有的数据存储到*/tmp/HBase-USERNAME*文件夹中。如果写访问此文件夹不可用，或者如果该磁盘已满，HBase将无法启动。使用以下命令，验证此目录中的文件是否可以创建：

```
mkdir  /tmp/HBase-$USERNAME
touch  /tmp/HBase-$USERNAME/test.txt
rm -rf  /tmp/HBase-$USERNAME
```

如果这三个命令中的任何一个命令失败，需验证该*/tmp*文件夹的权限。

环境变量

各类Hadoop和HBase相关的工具可能已经被测试，并可能需要定义一些环境变量。它们最有可能在HBase启动的时候引起问题。确保HADOOP_HOME和HBASE_HOME已经在你的环境中定义。使用下面的命令，列出所有Hadoop和HBase的相关环境变量：

```
export -p | egrep -i "(hadoop|HBase)"
```

unset命令可以用来去除这些变量，但这可能会对其他正在运行的应用程序产生影响。

可用内存

默认情况下，HBase会让JVM分配最大堆内存给所有HBase的进程。在一个内存为16 GB的机器中，JDK 1.7将允许一个Java进程使用多达4 GB内存。下面的命令将打印当没有特别指定内存参数的时候，分配多少内存给一个Java程序：

```
java -XX:+PrintFlagsFinal -version | grep MaxHeapSize
```

这适用单机模式的HBase进程，同样也适用以伪分布式模式或者全分布式模式运行的RegionServer和Master服务。不推荐将HBase的内存设置小于1 GB；确保你有足够的可用内存是很重要的，可以在*HBase-env.sh*文件中修改HBASE_HEAPSIZE变量来修改该限制，但该变量会影响HBase上所有的角色。

如果你只想修改特定角色的Java heap大小，选择需要配置的服务，添加Java -Xmx参数到HBASE_*service_OPTS*选项（例如，MASTER，REGIONSERVER，THRIFT，OOKEEPER或REST），其中的service代表你想要配置的服务。

不要在同一时间使用这些选项，因为它们可能会存在冲突。如果你决定用HBASE_service_OPTS方法修改每个进程的配置属性，然后一直注释HBASE_HEAPSIZE并指定每个进程所需的内存。如果决定使用HBASE_HEAPSIZE属性为所有服务分配相同数量的内存，就不要使用HBASE_service_OPTS方法。

在下面的例子中，我们分配2GB的内存给Master进程，8 GB的内存给RegionServer进程：

```
# export HBASE_HEAPSIZE=1000
export HBASE_MASTER_OPTS="$HBASE_MASTER_OPTS $HBASE_JMX_BASE -Xmx2g\
      -Dcom.sun.management.jmxremote.port=10101"
export HBASE_REGIONSERVER_OPTS="$HBASE_REGIONSERVER_OPTS $HBASE_JMX_BASE -Xmx8g\
      -Dcom.sun.management.jmxremote.port=10102"
```

确保你在分配给HBase至少1 GB的内存（然而请注意，我们为本书的例子推荐4 GB的内存）。

第一步

既然HBase正在运行，让我们做一些额外的步骤让工作更舒适。我们执行一些非常基本的HBase命令，下载并安装本书的一些实例，然后运行一些例子。所有在下面

的章节中描述的例子都可以通过本书的GitHub页面下载获得。一旦你掌握其中要点，你可以随时修改来看看不同的可选项的结果和影响。

基本操作

由于在整个章节都会使用HBase shell，我们回去做一些基本操作来熟悉它。如果你在任何时候发现自己摸不着头绪，你可以键入`Exit`或按Ctrl-C返回UNIX命令行或运行HBase shell 重启。在单机环境运行时，如果你的HBase实例进入一个糟糕的状态，你可以停止该进程，删除*/tmp/HBase-$USERNAME*文件夹，并重新启动。在虚拟环境中，你需要停止HBase，从HDFS删除*/HBase*目录的内容，从zookeeper删除/HBase节点，并重新启动。

下面的命令可能会帮助你实现：

```
hadoop fs -rm -r /HBase
echo "rmr /HBase" | grep HBase zkcli
```

最后一个命令需要zookeeper客户端。根据你环境安装的方式，客户端可能必须以不同的方式调用。

在本章结束部分我们对常用命令做些总结，创建一个表，然后列出当前所有的非系统表。

help

帮助命令列出了所有可用的HBase shell命令。在本书中，由于帮助命令的输出相当长，所以我们不会在这里重复它，但是，在接下来的几章中，我们将讨论大部分help中的命令。

create

`create`命令可以创建一个新的HBase表。下面的命令创建了一个列族名为`f1`，表名为`t1`的单列族表，所有的参数都是默认的：

```
hbase(main):002:0> create 't1', 'f1'
0 row(s) in 1.2860 seconds

=> HBase::Table - t1
```

List

`list`命令显示新创建的表：

```
hbase(main):003:0> list
TABLE
t1
1 row(s) in 0.0100 seconds

=> ["t1"]
hbase(main):004:0>
```

列表显示用户表。用户表不是系统表（即它们的表是由用户创建的）。

如果一切都已安装并正确运行，运行这三个命令，应该能够看到的输出信息与显示的内容相似。

代码导入实例

这本书中的例子都可通过本书的GitHub库获得。为简洁起见，重点关注关键部分的代码，这里介绍的例子已经做了浓缩。本书中使用的实例名称和GitHub库的文件名称是相匹配。包名对应于这本书的章节编号。这个命名约定将允许你轻松快速地在示例库中寻找到代码。例如，在例5-1中使用的`TestInstallation`示例代码，可以在名称*testinstallation.java*下找到*com.architecting.ch05*包。

此外，因为许多例子依赖于其他实例（例如扫描取决于表的创建和数据的生成），一些示例将重复使用其他相同的例子，一些辅助类用来准备环境以使结果可预测。你也可修改那些辅助类和例子来构建不同的环境并测试不同的场景，如生成更多的数据、更大的值，甚至生成被损坏的信息来测试故障处理。

为了下载、创建和修改实例，你将需要下列工具来安装和配置（如果你使用快速启动虚拟机运行这些实例，它上面已经安装了这些工具；如果使用一个Linux虚拟机运行实例，你将需要使用软件包管理工具如APT或yum安装）：

文本编辑器

为了查看示例并修改它们，你将需要一个文本编辑器。可以在终端安装XEmac或任何其他类型的编辑器，只要你觉得用起来舒服。为了简单起见，我们使用了Eclipse创建所有的例子和代码。

Git

正如其在网站上所说，“Git是一个免费开源的分布式版本控制系统，其设计着眼于高效快速的处理从小型到大型规模的项目。”你需要Git检索实例源代码。所有主流的Linux发行版本都能获得到Git。

本书的例子都托管在GitHub上，它提供了一个在线公共Git服务。你也可以使用Web浏览器访问和下载示例。

Maven

Apache Maven网站定义为“软件项目管理和理解工具”。基于项目对象模型（POM）的概念，Maven可以管理项目的构建，报告和文件信息的核心部分。HBase应用架构的源代码是通过Maven进行配置构建的。打包和编译的例子在下面的小节提供。

如果选择使用Cloudera快速启动VM 5.4.0或者更低版本进行开发，将需要使用命令行获得Git库。事实上，虚拟机提供的Eclipse版本不能和Eclipse m2e-egit plugin兼容。所以，如果喜欢在虚拟机中使用导入向导，将需要升级到一个更新的版本。

命令行下载

我们需要做的第一件事是下载源库。正如我们在上一节的提到的，代码实例托管在一个Git仓库中。如果你喜欢使用Eclipse完成所有这些操作，请参考本章后面的“使用Eclipse下载和构建”。

下面的Git命令将下载例子到本地文件夹：

```
git clone https://github.com/architectingHBase/examples.git ~/ ahae
```

这将创建一个目录名为*architecting-HBase-applications-examples*的例子，看起来像这样：

```
#:~/ahae$ ll
total 32K
drwxr-xr-x  2 jmspaggiari  jmspaggiari 4096   Sep 11 21:09 conf
-rw-r--r--  1 jmspaggiari  jmspaggiari 11324  Sep 11 21:09 LICENSE
-rwxr-xr-x  1 jmspaggiari  jmspaggiari 5938   Sep 11 21:09 pom.xml
-rw-r--r--  1 jmspaggiari  jmspaggiari 38     Sep 11 21:09 README.md
drwxr-xr-x  3 jmspaggiari  jmspaggiari 4096   Sep 11 21:09 src
```

如果你使用的是VM中的Eclipse，现在可以使用导入命令或Maven/Existing maven projects将项目导入到Eclipse中。

命令行构建

现在你已经获得源码，可以使用下列Maven命令构建项目：

```
#:~/ahae$ mvn package
```

这将下载所有所需的依赖的代码，将源代码编译成二进制文件，并将它们打包。你应该在你的控制台上看到这样的信息：

```
#:~/ahae$ mvn package
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building architecting-HBase-applications-examples 1
[INFO] ------------------------------------------------------------------------
Downloading: http://onejar-maven-plugin.googlecode.com/svn/mavenrepo/org/apache/\
maven/plugins/maven-resources-plugin/2.3/maven-resources-plugin-2.3.pom
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/\
maven-resources-plugin/2.3/maven-resources-plugin-2.3.pom
Downloaded: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/\
maven-resources-plugin/2.3/maven-resources-plugin-2.3.pom (5 KB at 18.1 KB/sec)
Downloading: http://onejar-maven-plugin.googlecode.com/svn/mavenrepo/org/apache/\
maven/plugins/maven-resources-plugin/2.3/maven-resources-plugin-2.3.jar
Downloading: http://repo.maven.apache.org/maven2/org/apache/maven/plugins/\
maven-resources-plugin/2.3/maven-resources-plugin-2.3.jar
.
.
.
[INFO] Building jar: /home/cloudera/ahae/target/ahae.jar
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 51.574s
[INFO] Finished at: Thu Sep 11 21:16:10 EDT 2014
[INFO] Final Memory: 40M/916M
[INFO] ------------------------------------------------------------------------
```

当你第一次运行构建命令时，其所耗时长取决于你的互联网连接的速度和要下载的依赖文件数量，这个操作可能需要几分钟。依赖的文件将被存储在本地Maven缓存文件夹，这将不必再下载除非它们改变。当完成时，你会在target目录下发现这个名字*ahae-1.jar*。这是我们将用来运行所有例子的文件。

使用Eclipse下载并构建

如果你想使用Eclipse来查看这些例子，你需要确保你有所需的m2e Eclipse Maven插件以及EGit连接器已经安装。Eclipse将允许你在一步一步的模式中，通过运行例子来检查不同的变量和结果。这是我们推荐运行例子的方式。

从文件菜单中选择导入选项，导入Java实例。Eclipse会提示你并列出它可以导入的所有不同种类的项目。Maven栏目下，选择“Check out Maven Projects from

SCM”，单击“Next”。此选项只会检查是否正确安装M2E插件，如果你Eclipse版本自带的插件已经安装。

如图5-1所示，在SCM URL项的下拉菜单中选择“git”选项，进入Git资料库URL的文本连接区：*https://github.com/architectingHBase/examples*。如果“git”选项不可用，请确保你已正确安装连接器。可以在同一窗口使用“m2e Marketplace”链接来安装连接器。当所有这些都完成后，只需单击“Finish”按钮，来导入project项目。这时窗口也将关闭，下载所有已定义的依赖关系到项目中，并在一个名为*architecting-HBase-applications-examples*应用实例中，导入源代码到你的Eclipse工作区。此操作可能需要几分钟，所以现在是一个很好的时间去抢一杯咖啡或去散步，你回来时，该项目应该已在你的工作区了。

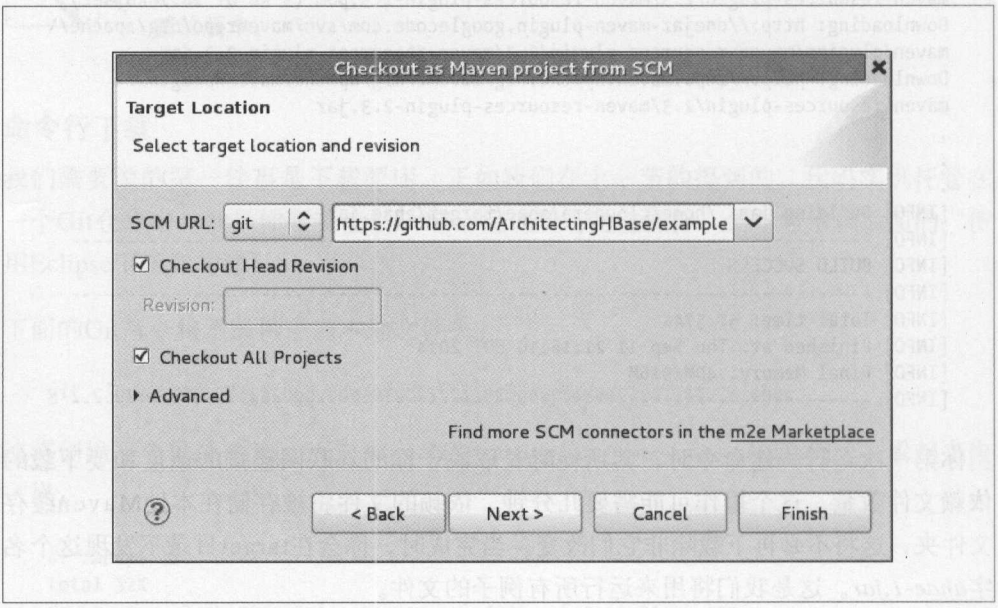


图5-1：Eclipse MVN导入窗口

我们可在Eclipse中使用Maven去构建我们的应用程序。这样的话，首先右键单击刚刚导入的项目，选择“Run As”选项，然后“Maven Build...”。此时会打开一个窗口来配置编译这个项目。只需要你修改的部分是“Goals”文本字段，在那里你需要输入“package”。一旦你完成了这些步骤，只需单击“Apply”按钮，然后单击“Run”按钮。在控制台的信息输出界面，你会看到Maven正在构建并打包你的例子。输出信息应该与本章前面的“命令行构建”相同。

为了简化所有的命令行，创建一个符号链接到你的主目录中的项目文件夹：

```
ln -s ~/workspace/architecting-HBase-applications-examples ~/ahae
```

测试实例

既然你已在HBase单机服务器中运行过实例，也已下载了所有实例，并且在你的系统中构建过本地应用，接下来我们将尝试运行一个简单的测试，以确保一切都正确配置。

HBase的配置信息存储在conf文件夹下的*HBase-site.xml*文件中。在单机模式运行时，这个文件可以是空的，所以你设置或者不设置这个文件的classpath都可以。然而，如果你修改这个文件，你需要确保它在classpath中可以被访问以便进行修改并被使用。你使用命令行还是Eclipse来运行你的例子将直接决定你是否需要将配置文件添加到classpath中。

命令行方式

当使用命令行运行实例时，你需要确保所有的HBase的二进制文件和配置文件都已经在classpath中。当你执行HBase `classpath`命令时，HBase将显示classpath的相关信息。另外例子中打包了二进制文件，正是通过这些路径，Java才能够正确运行这些实例。

下面的命令将从*com.architecting.ch05*中运行`TestInstallation`实例。

```
#:~/ahae$ java -classpath ~/ahae/target/ahae.jar:`HBase classpath`\
                                    com.architecting.ch05.TestInstallation
```

`TestInstallation`是一个简单的Java代码片段，试图调用HBase以检测其是否正在运行（参见例5-1）。

例5-1：使用TestInstallation检查HBase是否正在运行（java）

```
public class TestInstallation {
  private static final Log LOG = LogFactory.getLog(TestInstallation.class);

  public static void main(String[] args) {
    Configuration conf = HBaseConfiguration.create();
    try {
      LOG.info("Testing HBase connection...");
      HBaseAdmin.checkHBaseAvailable(conf);
      LOG.info("HBase is running correctly...");
    } catch (MasterNotRunningException e) {
      LOG.error("Unable to find a running HBase instance", e);
    } catch (ZooKeeperConnectionException e) {
      LOG.error("Unable to connect to ZooKeeper", e);
    } catch (ServiceException e) {
      LOG.error("HBase service unavailable", e);
```

```
    } catch (IOException e) {
      LOG.error("Error when trying to get HBase status", e);
    }
  }
}
```

这个命令调用HBase classpath命令获取当前的HBase classpath，然后追加实例JAR，并运行了JAR中的一个程序。这个命令会在不同的HBase类产生大量输出结果。你需要看的是从例子本身输出的信息，它应该看起来像下面的代码片段（注意，为了简洁起见，有些行已被删除或缩短；而同时，你可以在你的运行环境exprot calsspath变量，来避免需要对它的每个Java进程都指定变量）：

```
export CLASSPATH= ~/ahae/target/ahae.jar:`HBase classpath`

2014-09-14 INFO [main] ch05.TestInstallation: Testing HBase connection... ❶
2014-09-14 WARN [main] util.NativeCodeLoader: Unable to load native-hadoop\
    library for your platform... using builtin-java classes where applicable
2014-09-14 INFO [main] zk.RecoverableZooKeeper: Process identifier=hconnection-\
    0x783c342b connecting to ZooKeeper ensemble=localhost:2181
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:zookeeper.\
    version=3.4.6-1565, built on 02/20/2014 09:09 GMT
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:host.name=t430s
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:java.version=1.7.0_60
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:java.io.tmpdir=/tmp
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:java.compiler=<NA>
2014-09-14 INFO [main] zk.ZooKeeper: Client environment:os.name=Linux
2014-09-14 INFO [main] zk.ZooKeeper: Initiating client connection,\
    connectString=localhost:2181 sessionTimeout=90000 watcher=hconnection-\
    0x783c342b, quorum=localhost:2181, baseZNode=/HBase
2014-09-14 INFO [main-SendThread(localhost:2181)] zookeeper.ClientCnxn: Socket\
    connection established to localhost/0:0:0:0:0:0:0:1:2181, initiating session
2014-09-14 INFO [main] client.HConnectionManager$HConnectionImplementation:\
    Closing master protocol: MasterService
2014-09-14 INFO [main] client.HConnectionManager$HConnectionImplementation:\
    Closing zookeeper sessionid=0x1487467e621000a
2014-09-14 INFO [main] zk.ZooKeeper: Session: 0x1487467e621000a closed
2014-09-14 INFO [main-EventThread] zookeeper.ClientCnxn: EventThread shut down
2014-09-14 INFO [main] ch05.TestInstallation: HBase is running correctly... ❷
```

❶和❷是例子打印出来的两行结果。

如果你没有看到“HBase is running correctly...”这一行，你需要确保你的HBase服务器是否运行正常以及例子是否按照预期编译成功。

使用Eclipse

为了运行eclipse中的一个例子，你只需要右键单击它，选择“Run As”，然后单击“Java Application”。Eclipse运行TestInstallation的同一例子的输出结果应该更短，应该包括HBase输出日志。应该看到在控制台上打印以下几行：

```
2014-09-14 18:45:34 INFO ch05.TestInstallation: Testing HBase connection...
...
2014-09-14 18:45:35 INFO ch05.TestInstallation: HBase is running correctly...
```

当运行Eclipse示例时，*HBase-site.xml*配置文件默认包含在依赖中。你的本地配置文件的修改不会被Eclipse运行例子使用，除非你指定添加相应的HBase配置目录到你的项目中。可以在项目属性中这样做。在Java构建阶段，找到Libraries选项卡，单击“Add External Class Folder…”，将添加你的HBase-site.xml文件对应的目录。如果你是在单机模式下安装HBase，文件应该在*~/HBase/conf*下；否则，应该是在*/etc/HBase/conf/*目录下。我们还建议对Hadoop做相同的文件配置。

伪分布式模式和全分布式模式

在分布式模式运行和管理一个HBase集群，超出了本书的范围，这里不予讨论。为了更好地了解在这种模式下如何配置和运行HBase，建议借鉴Apache HBase的参考指南。但是，我们会在伪分布式模式下提供一些配置提示。

为了配置你的HBase实例来运行伪分布式或全分布式模式，在*conf/HBase-site.xml*文件中添加以下所需的配置参数：

```
<property>
  <name>HBase.cluster.distributed</name>
  <value>true</value>
  <description>The mode the cluster will be in. Possible values are
    false: standalone and pseudodistributed setups with managed ZooKeeper
    true: fully distributed with unmanaged ZooKeeper Quorum (see HBase-env.sh)
  </description>
</property>
```

除非你是单独运行zookeeper，否则你还需要告诉HBase来管理你的zookeeper。要做到这一点，打开*HBase/HBase-env.sh*文件并将以下条目更新为`true`：

```
# Tell HBase whether it should manage its own instance of ZooKeeper or not.
export HBASE_MANAGES_ZK=true
```

最后，在*HBase/HBase-env.sh*文件中，配置`JAVA_HOME`指向本地Java安装路径。

如果你在这些步骤中遇到任何问题，请参阅本章前面的“故障排除”或Apache HBase的参考指南。

第二部分

用例

在进入用例部分之前，我们需要对HBase的基本原理有一个充分的理解。在回顾完第一部分后，我们可以看看如何在现实生活的用例中使用HBase。接下来，我们将重点关注HBase的四个核心应用：HBase Solr，HBase的近实时服务系统，HBase主数据管理以及基于HBase的文件存储应用。

在整个第二部分中，我们将描述业务需求、数据输入方式、数据检索方式以及数据处理方式。其次，我们还将提供初步的解决方案以及相应用例的代码示例，这样可让你轻松了解HBase的应用价值。但凡进行具体方面的探讨时，我们也将详细介绍影响用例的方式及原因，必要时，会深入HBase的内部原理。

我们不可能在一本书中覆盖所有的HBase用例，所以选择了我们认为可以涵盖大部分HBase用法的用例。我们希望通过所有的描述场景，在一定程度上让你获得HBase的最佳实践，然后你就可以用同样的方式来解决你的具体用例。

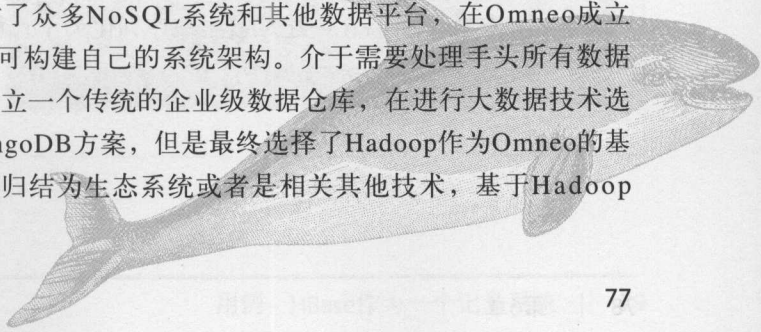
第6章

用例：HBase作为一个记录系统

我们将讨论的第一个用例来自于Omneo［西门子旗下Camstar(卡斯达)公司的一个分支］，Omneo是一个大数据分析平台，通过汇总不同来源的数据，为整条供应链的产品质量提供360度的数据视图。当前，各类规模的产品制造商都拥有着大量的数据，而且这些大量的，快速生成的多品种制造数据集包括了大批量的关键属性；当通过一个复杂的，多层次的供应链路去跟踪产品时，由于没有统一的数据格式，像这种集中式的数据存储将给我们提出了更加严峻的挑战。Omneo能够从整条供应链路的各个领域摄取数据，如制造、测试、装配、维修、服务和实地场景。

Omneo通过软件即服务（SaaS）的模式，为其终端用户提供平台服务，如调查产品质量问题，分析影响因素，确定项目的牵制和控制能力，以及为早期发现和纠正问题提供预留空间，从而可大大降低成本并显著提高了消费者的品牌信心。在这之中，Omneo首先通过关联数据源的方式构建一个统一的数据模型，以便用户可以在产品整个生命周期中分析影响产品质量的因素。此外，Omneo同时也提供了一个集中式的数据存储环境，能够在单一或统一的环境中促进所有产品质量数据的分析应用。

尽管它的一个拥有30多年行业经验的母公司Camstar也给了Omneo一个完善的 IT操作系统，但是Omneo还是评估了众多NoSQL系统和其他数据平台，在Omneo成立时，这个团队也被全权授予其可构建自己的系统架构。介于需要处理手头所有数据的艰巨任务，Omneo决定要建立一个传统的企业级数据仓库，在进行大数据技术选型时，考虑了Cassandra和MongoDB方案，但是最终选择了Hadoop作为Omneo的基础平台。这个决定的主要原因归结为生态系统或者是相关其他技术，基于Hadoop



能够提供如MapReduce，HBase，Solr和Impala等完全集成的生态系统，从而使得Omneo能够在一个单一的平台处理数据，而不需要在异构系统之间进行数据迁移。

该解决方案必须能够在一个集群中获取和处理大量的产品和客户数据，如果一个应用需要运行总数据的80%~90%，这样可能会使集群在处理数据时会变得相当的不稳定。Omneo在进行数据写入时，支持多用户在含50个节点的同一集群中处理超60亿原始数据处理，在HDFS中存储的数据约达100 TB，重要的是，纵观整个系统架构，我们会发现系统默认采用了数据备份机制，并引入了统一数据格式来进行数据存储。

Omneo采用了全面拥抱Hadoop生态系统的总体结构，并充分利用Hadoop的Avro数据序列化，在Hadoop生态系统中，avro也是最常用的文件存储格式，并且其允许采用同一模式来存储数据，从而使它更容易为不同的处理系统（例如，MapReduce、HBase和Impala/Hive）轻松访问数据而不需要多次的序列化和反序列化。

Omneo整体架构包含以下几个阶段：

- 摄取/预处理。
- 处理/服务。
- 用户体验。

摄取/预处理

摄取/预处理阶段包括平面文件的获取，HDFS文件的加载及Avro格式的转换，如图6-1所示，Omneo通过一个批处理方式获取所有文件，这些文件可为CSV文件格式或XML文件格式。然后通过HDFS API的方式将它们加载到HDFS中，并进行一系列的转换处理，最后将进行相关数据集的关联。其中，这些大多数的连接通常都是基于数据的主键（在电子制造的场景应用中，这通常是一个用于明确在整个生命周期的产品序列号）。另外，这一系列的转换基本都是通过MapReduce框架进行处理的。然而，Omneo也想通过一个图形化界面为客户提供数据整合服务，而不仅仅通过自定义的MapReduce代码。为了实现这一目标，Omneo与Pentaho合作加快产品研发，一旦数据被转换、关联在一起，就会被序列化为Avro格式。

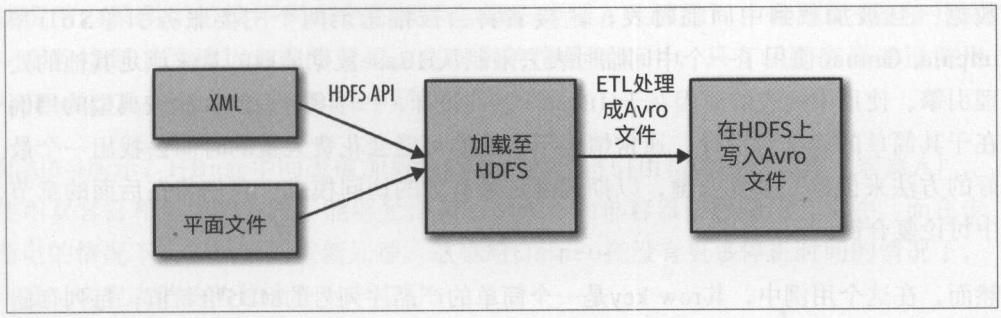


图6-1：HDFS API方式批处理摄取文件

处理/服务

一旦数据被转换成Avro 格式，它将被加载到HBase（见图6-2），因为数据已经被提交给Omneo批处理程序中，所以可在此基础上，采用批量加载工具，将数据加载到一个HBase临时表中。正如前面提到的MapReduce 任务输出的HFile文件，将最终加载到HBase目标库中，其中`completebulkload`工具首先传入文件在HDFS中URL路径；然后将每个file加载到RegionServer的相关region中。然而在HFile文件被加载后，region也会间歇性的触发分裂过程，批量加载工具会根据region的边界范围，自动的分裂出新的HFile文件。

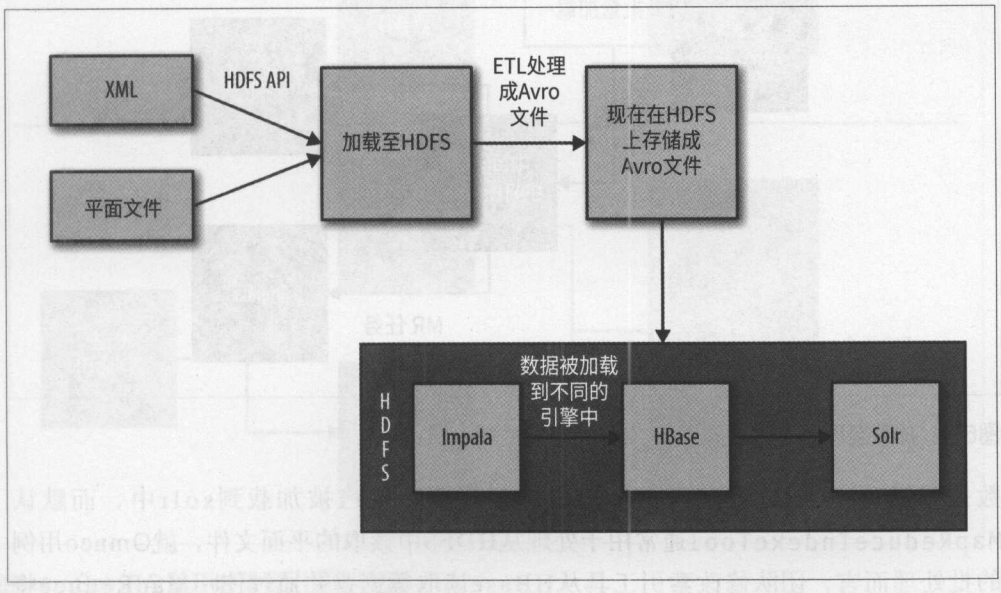


图6-2：采用统一的Avro存储格式

数据一旦被加载到中间临时表，紧接着将会被推送至两个主要服务引擎Solr和impala, Omneo使用了一个中间临时表去限制从HBase数据读取的量来满足其他的处理引擎。使用中间表的原因在于HBase key的设计，一个关于HBase比较典型的用例在于其简单的表结构设计，通常情况下，我们可能会花费大量的时间去找出一个最好的方法来创建一个复合键，以期望满足最有效的访问模式，我们将在后面的章节中讨论复合键。

然而，在这个用例中，其row key是一个简单的产品序列号的MD5哈希值，每列存储一条Avro记录，列名包含其存储的Avro记录的唯一ID，而Avro记录则是一个包含所有属性的非规范化数据集。

在数据被加载到HBase中间临时表后，继而会被传输到其他两个服务引擎。第一个服务引擎是Cloudera Search（Solr），其次是Impala，图6-3展示了整体数据加载入Solr的流程。

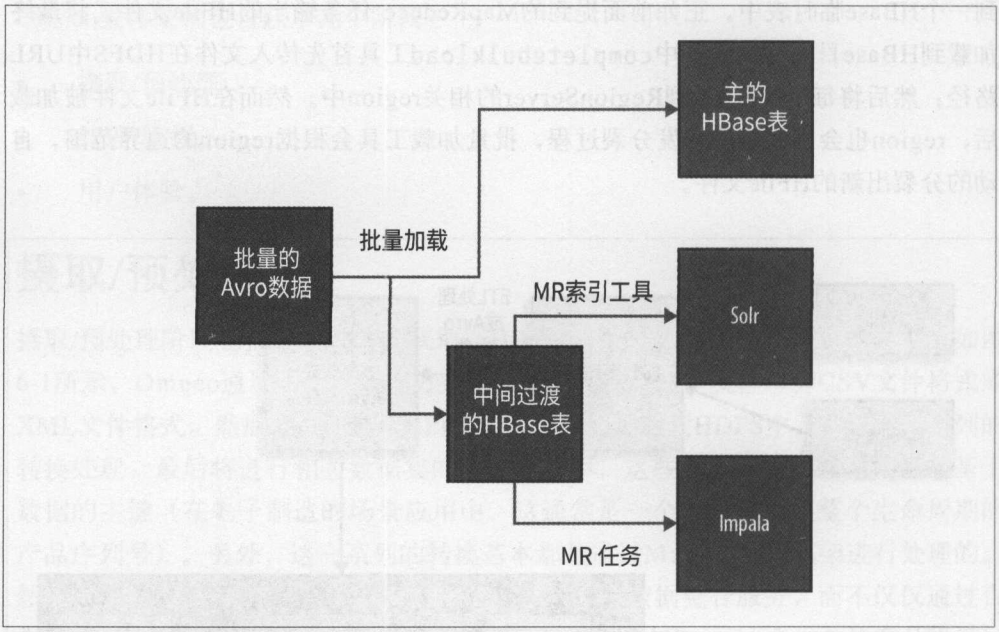


图6-3：加载至HBase的数据流

数据通过一个自定义的`MapReduceIndexeTool`被加载到solr中，而默认`MapReduceIndexeTool`通常用于处理从HDFS中读取的平面文件，就Omneo用例的批处理而言，团队修改索引工具从HBase读取数据，并通过使用MapReduce将数据直接写入Solr容器，图6-4展示了从HBase到Solr容器的两个数据流，在这种

情况下，采用了扮演不同角色的两个容器：容器A（活跃）、容器B（备份）和一个连接到“活跃”容器的引用，在索引的增量更新中，只有活跃容器通过使用MapReduceIndexerTool从HBase中间表获取更新。

如图6-4所示，HBase中间表被加载到容器，并且将引用指向活跃容器（容器A）。使用双容器和引用的方式，能够允许在一个单一的单容器中删除所有文件，而在未断电的情况下实现数据的重新处理，这也给Omneo在没有更多停机时间的情况下，可及时实现表结构的修改及生产应用。

图6-4的下半部分说明这样一种行为，MapReduceIndexerTool在重建HBase主表索引到容器B的时候，引用仍旧可指向容器A，一旦索引创建完成，引用可转向指向容器B，增量索引仍可指定到容器B直至该数据集又重新创建索引。

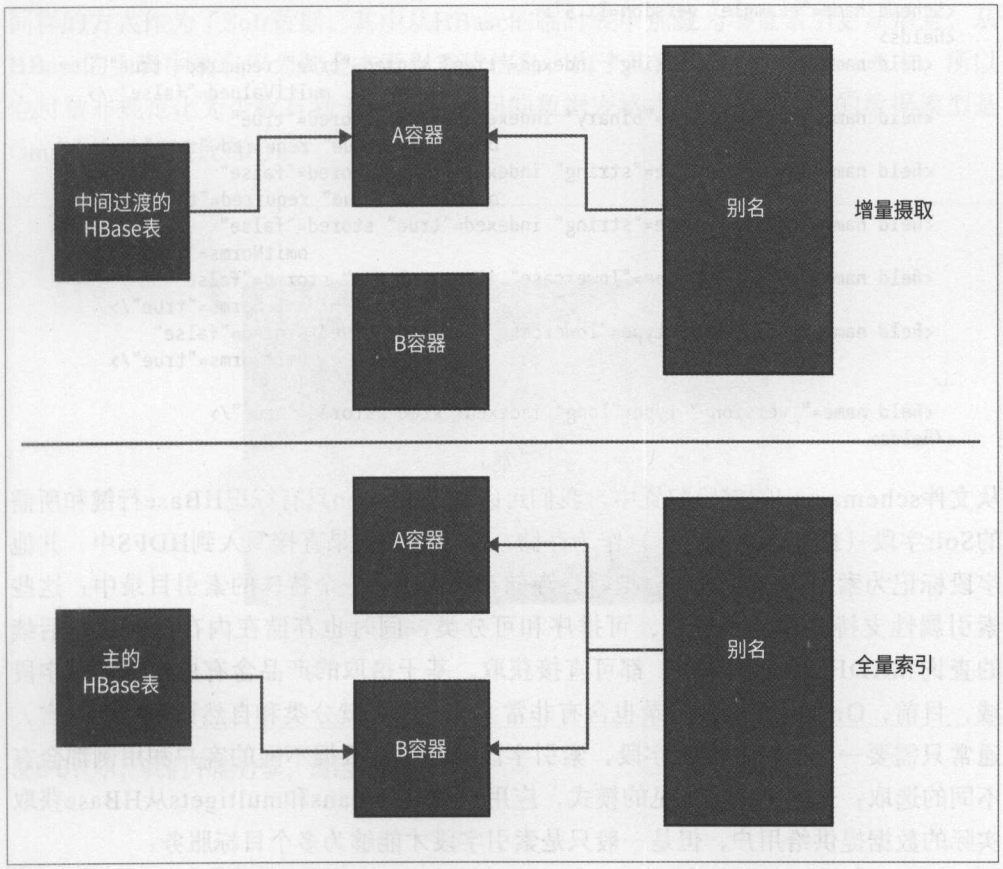


图6-4：Solr的全量及增量索引更新管理

这正是这个用例有趣的地方，HBase在整体架构中提供了两个主要功能，第一个用于处理主数据管理（MDM），在这种情况下，因为它允许更新，所以HBase可以看做是可用于Impala和Solr的记录系统，如果在Impala或Solr上碰到了数据集的问题，它们将可根据HBase的数据集实现重建，在HBase中，试图重新定义行键的结果通常意味着需要重建整个数据集，所以Omneo首先试图对二级、三级字段值域利用组合键的方式实现快速查找，但事实证明了，终端用户更加偏向于他们认定的通过改变主键的方式查找，例如，他们可能通过部分时间、测验、日期或任何其他字段进行数据查找。这就是为什么Omneo避免利用复合键，而采用通过Solr添加额外的索引到HBase的原因之一，其次，也是最重要的地方在于HBase中存储了终端用户所需要的所有记录，让我们一起看看Omneo 的Solr 配置文件schema.xml中部分字段域的配置实例：

```
<schema name="example" version="1.5">
<fields>
  <field name="id" type="string" indexed="true" stored="true" required="true"
                                                    multiValued="false" />
  <field name="rowkey" type="binary" indexed="false" stored="true"
                                    omitNorms="true" required="true"/>
  <field name="eventid" type="string" indexed="true" stored="false"
                                    omitNorms="true" required="true"/>
  <field name="docType" type="string" indexed="true" stored="false"
                                                    omitNorms="true"/>
  <field name="partName" type="lowercase" indexed="true" stored="false"
                                                    omitNorms="true"/>
  <field name="partNumber" type="lowercase" indexed="true" stored="false"
                                                    omitNorms="true"/>
...
  <field name="_version_" type="long" indexed="true" stored="true"/>
</fields>
```

从文件schema.xml的字段配置中，我们可以发现Omneo只有标记HBase行键和所需的Solr字段（ID和_version_）作为存储，并将这些结果直接写入到HDFS中，其他字段标记为索引；这些索引数据将会存储在HDFS的一个特殊的索引目录中；这些索引属性支持数据的可搜索、可排序和可分类，同时也存储在内存中，这样后续的查询和HDFS数据存储时，都可直接获取。基于摄取的产品含有成百上千的字段域，目前，Omneo摄取的记录也含有非常多的字段，就分类和自然语言检索而言，通常只需要一小部分必要的字段，索引字段的数量将根据不同的客户和用例都会有不同的选取；这是一个很常见的模式，应用程序调用scans和multigets从HBase获取实际的数据提供给用户，但是一般只是索引字段才能够为多个目标服务：

- 从Solr中提供全面的、更严格的、更多的可预测的SLAs服务将导致发生内存溢出异常。
- 现在的情况是构建在HDFS上的Solr云会将每个碎片及副本数据写入到HDFS，假如HDFS副本数量被设置为默认的3,这样一个有两个副本的碎片数据将会在HDFS上有9份拷贝数据，这通常不会影响搜索部署，因为内存或CPU通常在存储前才会造成瓶颈，但是这将会需要更多的存储空间。
- 字段检索所提供的快速统计机制，通常只需要统计已创建索引的字段，此功能可以帮助避免昂贵的SQL成本和基于HBase MapReduce预处理程序的汇总。

数据能够从HBase中以Avro格式加载进入Impala 表中，并同时能够转换为Parquet文件格式，Omneo采用Impala为其数据仓库服务提供给终端用户；同时这些数据也以同样的方式作为了Solr数据，其中从HBase的临时表中加载为增量索引更新信息，从HBase的主表中抽取作为其全文索引重建信息；由于其数据来源于HBase表中，所以它可被非规范化为少数有利于Impala访问的数据表格式，图 6-5采用的数据模型是Omneo的重要组成部分。

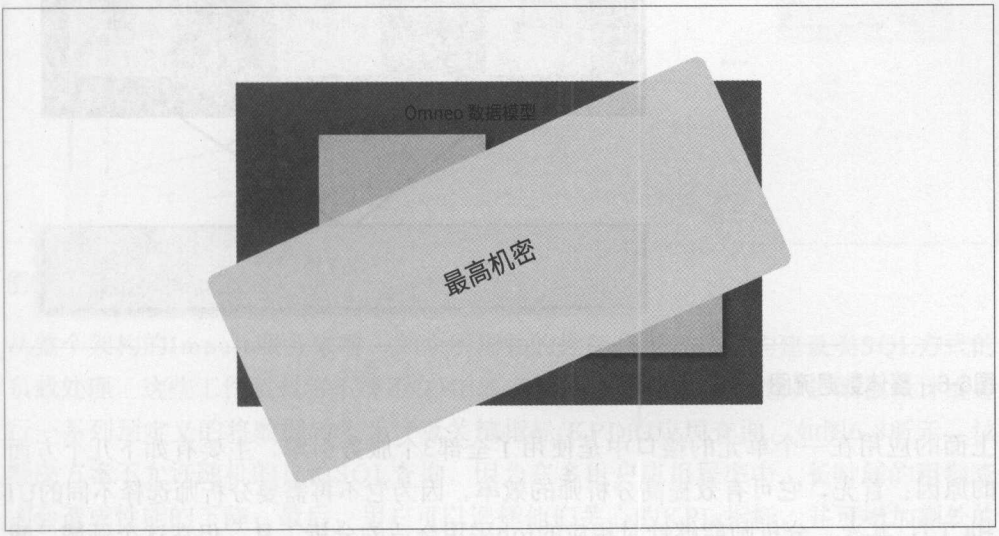


图6-5：不，我们不能分享。请自己领会!

用户体验

通常情况下，我们并不会花太多的时间在终端应用上，因为每个终端应用都千差万别，因此，在这种情况下，讨论如何将一切结合到一起的议题会变得更加重要，如图6-6所示，结合不同的引擎到一个高效的用户体验中将会是大数据的未来，这就是为什么很多公司从最初的玩大数据到真正提供一个巨大产品的原因。

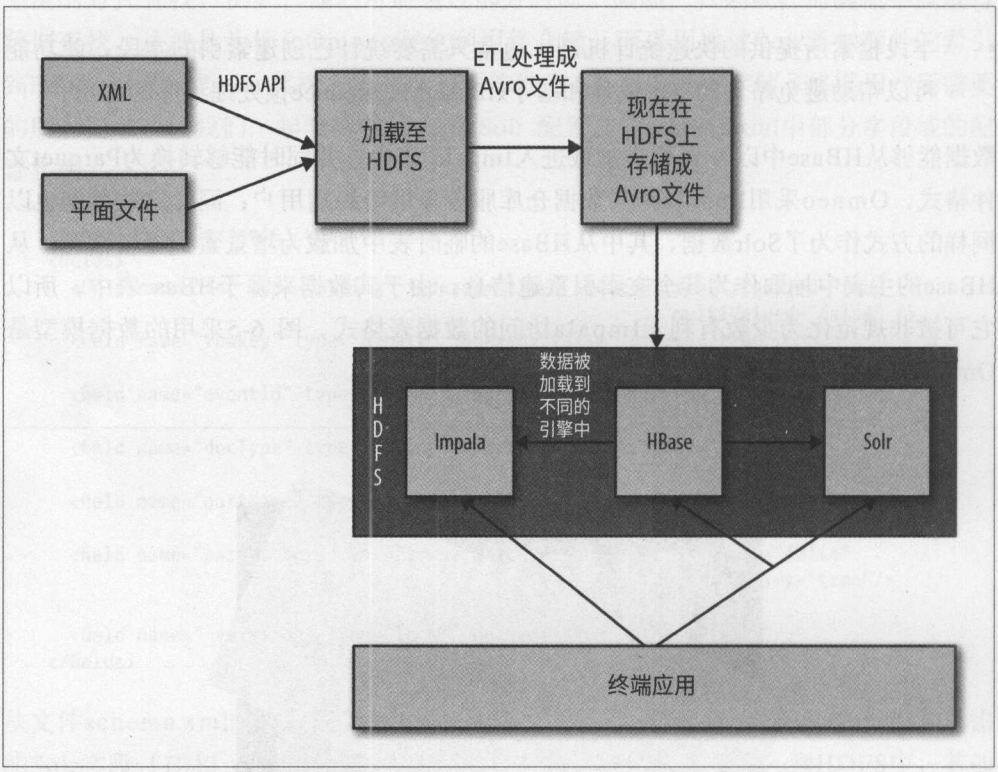


图6-6：整体数据流图，包括用户交互

上面的应用在一个单元的接口中是使用了全部3个服务引擎，主要有如下几个方面的原因：首先，它可有效提高分析师的效率，因为它不再需要分析师选择不同的UI和CLI；其次，分析师能够针对相应的job采用合适的分析工具，但在这个领域，我们可以发现一个重要的方面在于用户可能会尝试通过一个工具去解决所有的问题，通过让Solr提供事实服务，并处理自然语言搜索，HBase提供全部的记录存储服务，Impala则用于处理汇总及SQL查询，从而使得Omneo能够提供360度全方位的数据分析视图。

首先让我们从整个架构的Solr/HBase服务来看，这是Omneo应用中两个重要的交织服务，正如前面提到的，Solr存储了HBase真实的rowkey和绝大多数用户用于搜索及facet查询的索引字段，在该案例中，当用户向下钻取或增加一个新的facets(见图6-7),原生数据并不从Solr中获取，而是从HBase中使用Multiget排名前50的记录，从而使得分析师能够通过搜索和facets查询的方式获得真实的数据记录，并且也能够帮助分析师导出数据记录到平面文件；应用终端可以通过调用HBase 的scan 方式扫描数据表，并将结果信息写出到终端用户。

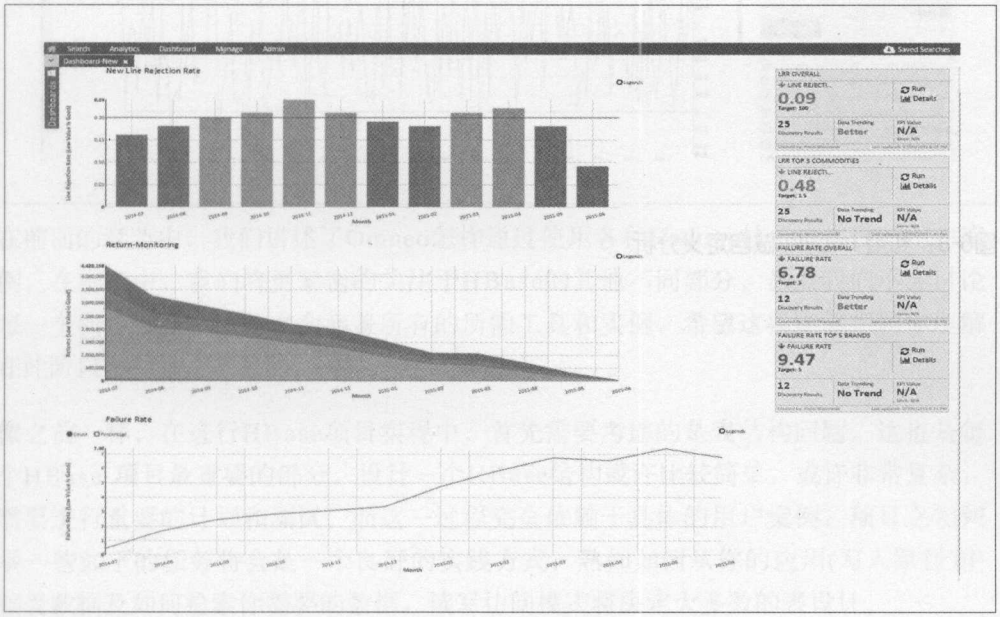


图6-7：看看Solr的优势

从整个架构的Impala服务来看，如众所周知的性能分析，模型构建及类SQL方式的负载处理，这些工作负载将不得不在HBase和Solr中体现，而性能分析则被设计成运行一系列预定义的将数据转化为计算关键指标(KPI)的应用查询，如图6-8所示，该解决方案不允许随机的自由SQL查询，因为在多租户应用程序中，长时间的粗糙查询会造成性能的下降；最后，用户可以选择他们关心的KPIs指标，并可增加额外的功能到查询中（sums，avg，max等）。

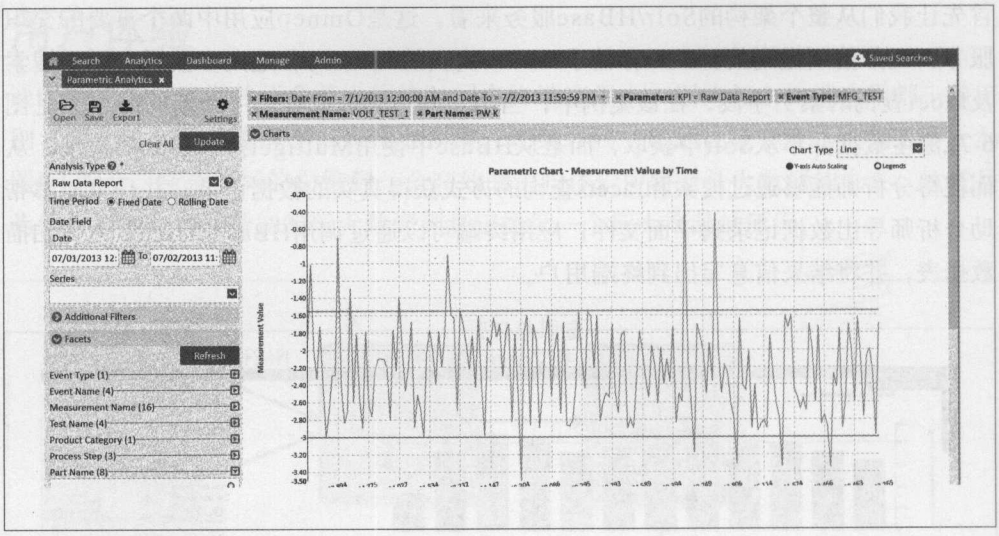


图6-8：利用 Impala 做自定义分析

第7章

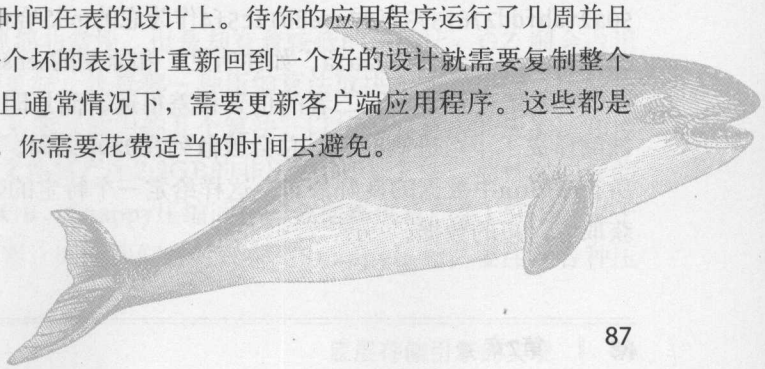
底层存储引擎的实现

在前面的章节中，我们讲述了Omneo怎样通过使用各种Hadoop技术去实现其用户案例，在本章中，我们将更紧密的关注于HBase的其他不同部分，虽然我们不会讨论每一个实施细节，但基本会涵盖所有的所需工具和实例，希望这样能够帮助你理解在此阶段中什么是重要的。

像之前一样，在进行HBase项目实现中，首先需要考虑的是表结构问题，这也是每个HBase 项目最重要的部分，设计一个HBase结构或许比较简单，或许非常复杂，需要进行重要的计划和测试，而这一过程完全依赖于具体的用户案例；项目之初列举一些如下的任务将会是一个良好的实践方式：熟知如何从你的应用(写入路径)中接受数据及如何检索你需要的数据，读写访问模式将决定大多数的表设计。

表设计

正如我们所说，我们也将继续在本书中不断提到，表设计是项目中最重要的部分之一。表结构，如你所选择使用的key以及你所配置的不同参数，这些不仅会对应用程序的性能产生影响，也会影响一致性问题。这也就是为什么对于所有我们将要讲解的用例，我们将会花费大量的时间在表的设计上。待你的应用程序运行了几周并且存储了TB级的数据之后，从一个坏的表设计重新回到一个好的设计就需要复制整个数据集，花费大量的时间，而且通常情况下，需要更新客户端应用程序。这些都是昂贵的操作，所以在这个阶段，你需要花费适当的时间去避免。



表结构设计

Omneo用例的表设计相当简单，不过我们还是要一步一步来，这样你就能将同样的方法应用到你自己的表结构设计中。我们读和写的路径都是高效的。在Omneo的案例中，数据是从外部系统批量获取的。因此，不同于其他的采集模式，数据每次插入单个值，这里数据可以直接以批次格式被处理而不需要单个的随机写入或者根据主键进行更新。在数据读取这方面，用户需要能够通过搜索sensor ID, event ID, date和event type的任意组合来快速检索一个特定sensor的所有信息。虽然我们无法设计一个能让所定检索标准都高效的key，但我们可以依赖于外部的索引，它会根据我们所定的标准返回一个用来查询HBase的key。我们可以简单的使用sensor ID的哈希值，同时将event ID 作为列限定符，因为这个key会从外部索引中被检索出来，所以我们并不需要查找或者扫描它。你可以参考本章后面的“生成测试数据”来查看数据格式的预览。

传感器可以拥有非常相似的ID，比如说42，43，44。然而，sensor ID也可以有一个很宽的范围（比如，40000~49000）。如果我们使用最初的sensor ID作为key，那么由于key的有序性，我们很可能会遇到特定区域的热点问题。关于热点问题的更多内容请阅读第16章。

哈希值

处理热点问题（hotspotting）的一种选择是根据这些已知的不同ID对表进行简单的预分区，以确保它们正确地分布在整个集群中。然而，如果这些ID的分布在未来变化了怎么办？在这种情况下，分区可能已经不正确了，我们可能又要面对某些region存在热点（hot spot）的问题。假设现在所有的ID 都在40××××和49××××之间，region将会被分成从开始到41，41到42，42到43等。但是如果明天一组新的sensor被加了进来，它们的ID从40×××到39×××，那么它们将会在第一个region结束。因为不可能预测未来的ID是什么样子，因此我们需要找到一个方法来确保无论ID是什么样子都会有一个好的分布。当对数据进行哈希时，即便开始时非常接近的key也会产生一个完全不同的结果。在这个例子中，42将会产生`50a2fabfdd276f573ff97ace8b11c5f4`作为它的md5哈希值，而43 将会产生`f0287f33eba7192e2a9c6a14f829aa1a`。如你所见，和原始的sensor ID值42和43不同，对这两个md5哈希值进行排序，二者的位置相差很远。而且即使新的ID值进来了，因为它们被翻译成了十六进制的值，总会被散列在0和F之间。使用这种hash的方法可以确保所有region中数据的良好分布，这样给定一个特定的sensor ID，我们仍然能够直接获取它对应的数据。

当你需要扫描数据时，而且也需要保持key的原有顺序不变的时候，哈希方法不能使用。因为key的md5值会破坏原有的顺序。

列限定符

关于列限定符，将会使用event ID。event ID是一个来自下游系统的哈希值，它对于特定传感器的给定事件是唯一的。每个时间都有特定的类型，诸如“警报”，“警告”或者“RMA”（return merchandise authorization，退货授权）。起初，我们考虑使用事件类型（event type）作为列分隔符。然而，一个传感器可以多次产生单个事件类型。传感器产生的每个“警告”会覆盖之前的“警告”，除非我们使用HBase的“版本”特性。使用唯一的事件ID（event ID）作为列分隔符允许我们对于同一个传感器存储多个相同类型的事件，且不需要编写额外的逻辑代码来使用HBase的“版本”特性，进而获取一个传感器的所有事件。

表参数

为了获得最佳性能，我们必须检查所有的参数并且确保它们是根据我们的需要和用途而被设定为相应的值。不过在这里，我们仅列出了应用于此特定用例的参数。

压缩

我们要检查的第一个参数是当将表中数据写到磁盘的时候所使用的压缩算法。HBase是以块格式将数据写进HFile。每个块默认是64 KB，并且是没有压缩的。块内存储的数据属于一个region和列族。通常情况下，一个表的列都包含有相关的信息，这些信息构成了一个共同的数据模式。压缩这些块总是能起到很好的效果。例如，压缩包含日志信息和客户信息的列族是一个不错的选择。HBase支持大多数压缩算法：LZO，GZ (对于 GZip)， SNAPPY和 LZ4。每一个压缩算法都有自己的优缺点。对于每种算法，都会权衡压缩和解压缩操作的压缩比对性能的影响（即在保证压缩算法正常运行的情况下，数据能否被充分压缩）。

Snappy在几乎所有操作中表现都非常快，但是却有着较低的压缩比；GZ 则会占用更多的资源，但通常会压缩的更好。选择哪一种压缩算法取决于你的用例。比较推荐的做法是用一个样例数据集来测试其中的几个算法，以检验各算法的压缩比和性能。例如，一个1.6GB的CSV文件将产生2.2GB的非压缩HFile文件，而针对同样的数据集，使用LZ4只会产生1.5GB。Snappy压缩同样的数据集也是产生1.5GB。由于读写的延迟对我们而言非常重要，因此我们将对表采用Snappy压缩。要注意各种压

缩算法在不同Linux发行版中的可用性。例如，Debian 默认不支持Snappy库。由于许可证问题，LZO和LZ4 库通常情况下也不和Apache Hadoop发行版绑定，必须单独安装。

需要注意的是根据数据类型的不同，压缩比可能不一样。实际上，压缩一个文本文件（text file），将比一个PNG格式的图片压缩的更好。例如，一个143976字节的PNG文件将只会被压缩到143812字节（只节省了2.3%的空间），而一个143509字节的XML文件则能被压缩到6284字节（节省了95.7%的空间）。推荐大家在选择一个压缩算法之前，在你的数据集上对不同的算法做个测试。如果压缩比不明显的话，应该避免使用压缩从而减轻处理器的负担。

数据块编码

数据块编码是HBase的一个特性，即key会根据前一个key进行编码和压缩。其中一个编码选项（FAST_DIFF）让HBase只存储当前key和前一个key不相同的地方。HBase独立存储每个单元（cell），包括它的key和value。当一行有多个cell时，为每个cell写入相同的key将会消耗大量的空间。因此，启动数据块编码可以节省大量空间。多数情况下，启动数据块编码都是有用的，所以，如果你不确定的话，就启动FAST_DIFF吧。当前的用例将会从这个编码中受益，因为一个给定的行中可以有上千列。

布隆过滤器

布隆过滤器可以跳过来自HBase region中的输入文件，从而有效的减少不必要的I/O。布隆过滤器会告诉HBase一个给定的key可能是或者不是在给定的文件中。但是，这并不意味着这个key绝对包含在此文件中。

不过，在某些特定的情况下布隆过滤器是不需要的。针对目前的用例而言，文件每天下载一次，之后表上会运行一个大合并。结果是，每个region几乎都是只有单个文件。而且，针对HBase 表的查询会由Solr返回的结果确定。这意味着读请求总是会成功并返回值。鉴于此，布隆过滤器总是会返回true并且HBase也将总能打开文件。因此，针对这个用例而言，布隆过滤器是一个额外开销并且是不需要的。

因为布隆过滤器默认是开启的，在这个案例中我们将需要明确的禁用它。

预分区

预分区并不是表参数。预分区信息并不和表的元数据信息一起存储，而且只在表创建的时候使用。不过，在继续我们的实现之前，理解这一步是非常重要的。对一个表进行预分区意味着让HBase在表创建的时候将它分割成多个region。HBase自带有很多不同的预分区算法。对一个表进行预分区的目的是确保初始化加载的数据会被正确的分散到各个region，而且不会在产生单个region的热点问题。诚然，随着时间推移，数据将会在region分割自动触发的时候被分散，但预分区将分散的动作提前至起始的时候。

实现

既然我们决定了为我们的表设置什么样的参数，那就是时候创建它了。我们将会保留所有默认的参数，除了之前我们讨论过的那些。在HBase shell中运行以下命令来创建一个叫做“sensors”的表，它含有一个列族以及我们讨论过的参数，预分区成15个region（NUMREGIONS和SPLITALGO这两个参数用来帮助HBase对表进行预分区）：

```
Hbase(main):001:0> create 'sensors', {NUMREGIONS => 15,\
                                      SPLITALGO => 'HexStringSplit'}, \
                                     {NAME => 'v', COMPRESSION => 'SNAPPY',\
                                      BLOOMFILTER => 'NONE',\
                                      DATA_BLOCK_ENCODING => 'FAST_DIFF'}
```

当你的表已经创建完成，就可以使用HBase WebUI接口或者下面的shell命令来查看它的详细信息：

```
Hbase(main):002:0> describe 'sensors'
Table sensors is ENABLED
sensors
COLUMN FAMILIES DESCRIPTION
{NAME => 'v', DATA_BLOCK_ENCODING => 'FAST_DIFF', BLOOMFILTER => 'NONE',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'SNAPPY',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.1410 seconds
```

NUMREGIONS和SPLITALGO这两个参数在表被创建的时候使用，但它们并不存储在表的元数据信息中。在表已经被创建之后再去检索这些信息是不可能的。

如你所见，我们所指定的参数已经列在输出中了，当然同时列出的还有默认的参数。默认参数由于你所使用的HBase版本的不同可能会有不同。不过BLLOMFILTER，DATA_BLOCK_ENCODING和COMMPRESSION需要按照我们指定的值来配置。

既然我们的表已经准备好了，可以进一步开始着手准备数据了。

数据转换

为了实现并测试我们所描述的用例，我们需要将数据导入到系统中。因此，需要产生一些测试数据以便后续的处理和转换操作。

生成测试数据

下一步我们的目的是产生一个具有代表性的测试数据集，使用对应用程序来处理该数据集并检验结果。首先，我们将会创建一些带有测试值的数据文件，在此基础上，你可以运行不同的命令和程序。

在这个例子中，你将会发现一个叫做CSVGenerator的类，它所产生的数据类似下面这样：

```
1b87,58f67b33-5264-456e-938a-9d2e9c5f4db8,ALERT,NE-565,0-0000-000,1,ECEGYFFL ...
3244,350cee9e-55fc-409d-b389-6780a8af9e76,RETURNED,NE-382,0-0000-000,1,OOQTY ...
727d,b97df483-f0bd-4f24-8ff3-6988d8eff88c,ALERT,NE-858,0-0000-000,1,MSWOCQXM ...
53d4,d8c39bf8-6f5f-4311-8ee5-9d3bce3e18d7,RETURNED,NE-881,0-0000-000,1,PMKMW ...
1fa8,4a0bf5b3-680d-4b87-8d9e-e55f06614ae4,ALERT,NE-523,0-0000-000,1,IYIZSHKA ...
```

每一行包含一个随机的且是由四个字符（0到65535，十六进制表示）组成的sensor ID，之后是一个随机的event ID，document type，part name，part number，version和由随机字母（664至128个字符长度）组成的payload。想要产生不同的数据集，你可以在任何你想要的时候重新运行CSVGenerator代码。后续部分的示例代码将会从*~/ahae/resources/ch07*文件夹中读取文件。这个类创建文件和它在哪里运行有关系，因此我们需要在*~/ahae*文件夹下运行这个类。如果你想增加或者减少数据集的大小，很简单，只需要更新下面这一行：

```
for (int index = 0; index < 1000000; index++) {
```

你可以直接在eclipse中运行这个数据产生器而不需要任何参数，或者在shell中的*~/ahae*目录下使用如下命令：

```
HBase -classpath ~/ahae/target/ahae.jar com.architecting.ch07.CSVGenerator
```

此命令将会在*~/ahae/resources/ch07/omneo.csv*中产生一个叫做*omneo.csv*的文件。

创建Avro文件格式

现在已经有了一些数据，我们需要定义一个能够映射该数据格式的Avro文件结构。根据在之前章节所提供的查找模式，我们需要如下格式的Avro文件结构：

```
{ "namespace": "com.architecting.ch07",
  "type": "record",
  "name": "Event",
  "fields": [
     {"name": "id", "type": "string"},
     {"name": "eventid", "type": "string"},
     {"name": "docType", "type": "string"},
     {"name": "partName", "type": "string"},
     {"name": "partNumber", "type": "string"},
     {"name": "version", "type": "long"},
     {"name": "payload", "type": "string"}
  ]
}
```

你可以发现，*omneo.avsc*文件就是采用了这种结构，此文件在*resources/ch07*目录中。因为它已经被编译并导入工程中，因此不需要重新编译。但是如果你想修改它，你可以使用如下命令编译它：

```
java -jar ~/ahae/lib/avro-tools-1.7.7.jar compile schema omneo.avsc ~/ahae/src/
```

此命令将创建一个包含Event对象的文件，即*~/ahae/src/com/architecting/ch07/Event.java*，它将被用来存储Event的Avro对象至HBase中。

实现MapReduce的转换

正如例7-1所示，生产处理的第一步是解析收到的CSV文件来产生HBase的HFile，这些文件将作为下一步的输入，它们会对应之前已创建的表的格式。

我们的生产数据将是非常巨大的文件，因此将使用MapReduce来实现，以充分利用并行的优势。这个MapReduce作业的输入将是text file，而输出则是HFile。这决定了你配置MapReduce作业的方式。

例7-1：转换成HFile

```
Table table = connection.getTable(tableName);
Job job = Job.getInstance(conf, "ConvertToHFiles: Convert CSV to HFiles");
HFileOutputFormat2.configureIncrementalLoad(job, table,
                                connection.getRegionLocator(tableName)); ❶
job.setInputFormatClass(TextInputFormat.class); ❷
```

```
job.setJarByClass(ConvertToHFiles.class); ❸
job.setJar("/home/cloudera/ahae/target/ahae.jar");❸
job.setMapperClass(ConvertToHFilesMapper.class); ❹
job.setMapOutputKeyClass(ImmutableBytesWritable.class); ❺
job.setMapOutputValueClass(KeyValue.class); ❻
FileInputFormat.setInputPaths(job, inputPath);
HFileOutputFormat2.setOutputPath(job, new Path(outputPath));
```

❶ HBase提供了一个帮助类来帮助你完成大多数配置。当你想配置你的MapReduce作业来产生HFile作为输出时，这是第一步需要调用的。

❷ 这里我们想读取一个带有CSV数据的text file，因此将使用`TextInputFormat`。

❸ 当从命令行运行的时候，所有需要用到的类都绑定到了一个客户端JAR上，它由`setJarByClass`方法指定。而如果从eclipse运行的话，就必须手动指定JAR路径，因为我们运行的类来自Eclipse环境，而MapReduce是不知道的。鉴于此，我们需要向MapReduce提供外部文件的路径，当然此路径中的类必须是可用的。

❹ 定义你用来解析CSV内容并创建Avro输出想要使用的mapper。

❺ 我们需要定义`ImmutableBytesWritable`作为mapper输出key的类。这将是我们用来写key的格式。

❻ 我们需要定义`KeyValue`作为mapper输出value的类。这代表着我们想要存入HFile的数据。

用来创建HFile的reducer在能够将单行数据的所有列写入HFile之前，需要将它们加载至内存并排序。如果你的数据集中有非常多的列，有可能内存溢出，这个问题将在未来的某个版本中修复，HBASE-13897中将会实现。

Mapper端的操作非常简单。目的只是将一行分割成不同的部分，将它们分配给一个Avro对象，并将这个Avro对象存储在HFile文件中，随时加载至HBase框架中。

正如例7-2所示，首先要做的是定义一系列的变量，我们将在每个mapper的每一次迭代中重用这些变量。这么做是为了减少创建对象的数量。

例7-2：转换成HFile的mapper

```
public static final ByteArrayOutputStream out = new ByteArrayOutputStream();
public static final DatumWriter<Event> writer = new SpecificDatumWriter<Event>
                                                    (Event.getClassSchema());

public static final BinaryEncoder encoder = encoderFactory.binaryEncoder(out,null);
public static final Event event = new Event();
public static final ImmutableBytesWritable rowKey = new ImmutableBytesWritable();
```

这些对象全部在例7-3 中的map方法中被重用。

例7-3：转换成HFile的mapper

```
//从接收的行中提取不同的字段
String[] line = value.toString().split(",");❶

event.setId(line[0]);
event.setEventId(line[1]);
event.setDocType(line[2]);
event.setPartName(line[3]);
event.setPartNumber(line[4]);
event.setVersion(Long.parseLong(line[5]));
event.setPayload(line[6]); ❷

//序列化AVRO 对象为ByteArray字节数组
out.reset(); ❸
writer.write(event, encoder); ❹
encoder.flush();

byte[] rowKeyBytes = DigestUtils.md5(line[0]);
rowKey.set(rowKeyBytes); ❺
context.getCounter("Convert", line[2]).increment(1);

KeyValue kv = new KeyValue(rowKeyBytes,
                        CF,
                        Bytes.toBytes(line[1]),
                        out.toByteArray());❻
context.write (rowKey, kv); ❼
```

❶ 首先，我们将一列分割成几部分，这样就能够直接地单独处理每部分。

❷ 我们为所有的map调用重用同样的对象，只是简单的分配给它新接收的值，而不是在每次迭代时创建一个新的Avro对象。

❸ 这是对象重用的另一个例子。在你的mapper代码中创建的对象越少，你需要的垃圾回收就越少，并且你的程序执行就越快。每个输入文件的每一行都会调用map方法。创建一个ByteArrayOutputStream并针对每个map迭代重用它和它的内部缓存，将会节省数百万计的对象创建。

❹ 序列化Avro对象，并将其转换为HBase可接受的字节数组，尽可能的重用已存在的对象。

❺ 利用sensor ID构造HBase的key。

❻ 利用我们的key和列族构造HBase的keyvalue。我们将event ID作为列限定符，Avro对象作为值（value）。

❼ 分发keyvalue对象，这样reducer就可以重组它们并写入需要的HFile中。行键（row key）只被用来分割数据。当数据将被写入基础文件时，只有KeyValue数据会被同时用作key和value。

当实现一个MapReduce作业时，在不必要的时候避免创建对象。如果你需要使用一个字符串的一小部分，不推荐使用split()方法来提取所需部分。在一个包含一千万字符串，每个字符串由50个部分组成的数据集上，使用split()方法将会产生5亿个对象，它们都需要垃圾回收。相反，解析字符串来找到所需要的那部分的位置并使用substring()方法。同时也可考虑使用来自Guava库的com.google.common.base.Splitter对象。

这个例子可以直接在Eclipse中运行，也可以在命令行中执行。无论在哪里执行，你都需要指定输入文件，输出目录和表名称作为参数。表名称是必需的，HBase用它来找到region的边界以便在输出数据中创建所需的分区。同时，也可用来查找对应的列族参数，比如压缩算法以及编码格式等参数。MapReduce作业将会根据表的region和列族参数在输出目录中产生HFile。

接下来的命令将会在HDFS上创建HFile（假如你在standalone模式下运行，你需要在本地磁盘上产生这些文件，只需更新目标文件夹即可）：

```
HBase -classpath ~/ahae/target/ahae.jar:`HBase classpath` \
com.architecting.ch09.ConvertToHFiles \❶
file:///home/cloudera/ahae/resources/ch09/omneo.csv \❷
hdfs://localhost/user/cloudera/ch09/hfiles/ sensors❸
```

❶ 调用转换类。

❷ 我们的输入文件。

❸ 输出文件夹及表名称。

如果你从Eclipse中开始此类，确保你已经从导航栏（运行→运行配置/参数）添加了参数。

因为这将启动一个MapReduce作业，所以输出会非常冗长并且会给出很多信息。注意以下几行：

```
Map-Reduce Framework
        Map input records=1000000
        Map output records=1000000
        Reduce input groups=65536
```

这里Map input records的值代表你CSV文件的行数。由于针对每一行我们只产生一个Avro对象，所以它的值和Map output records的值相匹配。Reduce input groups的值代表唯一key的数量。因此这里我们能够看到，原文件一百万行对应有65536行不同的数据，这告诉我们每行大约有15列。

在此处理过程的最后，你文件夹中的内容应该看起来像下面这个样子：

```
[cloudera@quickstart ~]$ hadoop fs -ls -R ch07/
drwxr-xr-x     0 2015-05-08 19:23 ch07/hfiles
-rw-r--r--     0 2015-05-08 19:23 ch07/hfiles/_SUCCESS
drwxr-xr-x     0 2015-05-08 19:23 ch07/hfiles/v
-rw-r--r-- 10480 2015-05-18 19:57 ch07/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e
-rw-r--r-- 10475 2015-05-18 19:56 ch07/hfiles/v/46d20246053042bb86163cbd3f9cd5fe
-rw-r--r-- 10481 2015-05-18 19:56 ch07/hfiles/v/6419434351d24624ae9a49c51860c80a
-rw-r--r-- 10468 2015-05-18 19:57 ch07/hfiles/v/680f817240c94f9c83f6e9f720e503e1
-rw-r--r-- 10409 2015-05-18 19:58 ch07/hfiles/v/69f6de3c5aa24872943a7907dcabba8f
-rw-r--r-- 10502 2015-05-18 19:56 ch07/hfiles/v/75a255632b44420a8462773624c30f45
-rw-r--r-- 10401 2015-05-18 19:56 ch07/hfiles/v/7c4125bfa37740ab911ce37069517a36
-rw-r--r-- 10441 2015-05-18 19:57 ch07/hfiles/v/9accdf87a00d4fd68b30ebf9d7fa3827
-rw-r--r-- 10584 2015-05-18 19:58 ch07/hfiles/v/9ee5c28cf8e1460c8872f9048577dace
-rw-r--r-- 10434 2015-05-18 19:57 ch07/hfiles/v/c0adc6cfceef49f9b1401d5d03226c12
-rw-r--r-- 10460 2015-05-18 19:57 ch07/hfiles/v/c0c9e4483988476ab23b991496d8c0d5
-rw-r--r-- 10481 2015-05-18 19:58 ch07/hfiles/v/ccb61f16feb24b4c9502b9523f1b02fe
-rw-r--r-- 10586 2015-05-18 19:56 ch07/hfiles/v/d39aeea4377c4d76a43369eb15a22bff
-rw-r--r-- 10438 2015-05-18 19:57 ch07/hfiles/v/d3b4efbec7f140d1b2dc20a589f7a507
-rw-r--r-- 10483 2015-05-18 19:56 ch07/hfiles/v/ed40f94ee09b434ea1c55538e0632837
```

为了适应页面显示，这里我们减掉了属主和组信息。所有的文件都属于启动MapReduce作业的用户。

如你在文件系统中所见，MapReduce作业创建的HFile文件个数和表中region的数量是一样的。

当产生输入文件时，一定要注意提供正确的列族。事实上，没有给MapReduce作业提供正确的列族名称是一个常见的错误，目录结构是根据列族名称来创建的，这将导致批量加载阶段作业失败。

在给出的例子中，存储文件的目录名称是根据我们在代码中“v”指定的列族名称来命名的。

HFile校验

整个处理过程中，我们在控制台得到的所有信息都和MapReduce架构及相应的运行任务相关。不过，尽管它们成功了，它们所产生的内容也可能是不对的。例如，我们可能用错了列族名称，在建表的时候忘记配置压缩算法，或者其他的某些错误。

HBase自带一个读取HFile并提取元数据信息的工具。此工具叫做HFilePrettyPrinter，它可以使用如下命令调用：

```
HBase hfile -printmeta -f ch07/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e
```

这个工具需要的参数只有HFile在HDFS中的存储位置。

这里我们展示部分之前命令的输出（某些和本章无关的那部分信息被省略了）：

```
Block index size as per heapsize: 161264
reader=ch07/hfiles/v/345c5c462c6e4ff6875c3185ec84c48e,
   compression=snappy, ❶
   cacheConf=CacheConfig:disabled,
   firstKey=7778/v:03afef80-7918-4a46-a903-f6e35b629926/1432004229936/Put, ❷
   lastKey=8888/v:fc69a89f-4a78-4e2d-ae0a-b22dc93c962c/1432004229936/Put, ❸
   avgKeyLen=53, ❹
   avgValueLen=171, ❺
   entries=666591, ❻
   length=104861200 ❼
```

现在让我们一起看一下这些输出的重要部分。

❶ 这里向你展示了你的文件所使用的压缩格式，这和你创建表的时候所做的配置相对应（我们初始时选择了Snappy，但是如果你配置了一个不同的，你应该在这里看到它）。

❷ 这个HFile的第一个cell的key，也包含了列族的名称。

❸ 此HFile中包含的最后一个key（只有key值在7778和8888之间的key才在此文件中）；当你要找的key不在第一个key和最后一个key之间的时候，它被HBase用来跳过整个文件。

❹ key的平均大小。

❺ value的平均值。

❻ HFile中cell的数量。

❼ 整个HFile的大小。

利用这个命令的输出，你能够校验文件中你所产生的数据以及数据的格式是否符合你的预期（压缩，布隆过滤，平均key大小等）。

批量加载

批量加载将预生成的HFile插入HBase中，而不是使用HBase API一个接一个的加载。批量加载是将大量值插入系统的最快速高效的方式。这里我将给大家展示是如何做到批量加载的。

你的表在HDFS中的内容应该看起来像下面这样（为了适应页面的宽度，文件权限信息和属主信息被删除了，并且/HBase/data/default/sensors被简写成.../s）：

```
  0 2015-05-18 19:46 .../s/.tabledesc
287 2015-05-18 19:46 .../s/.tabledesc/.tableinfo.0000000001
  0 2015-05-18 19:46 .../s/.tmp
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd
 58 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/.regioninfo
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/recovered.edits
  0 2015-05-18 19:46 .../s/0cc853926...3d12959bbcacc55fd/recovered.edits/2.seqid
  0 2015-05-18 19:46 .../s/0cc853926c7c10d3d12959bbcacc55fd/v
```

如果你的表是空的，你仍然会看到所有region的文件夹，因为我们对表进行了预分区。如果在加载之前数据已经存在了，HFile就可能存在于region文件夹中。在之前的片段中我们只展示了一个region的目录，你可以看到这个region的列族v是空的，因为它没有包含任何的HFile文件。

我们的HFile已经由MapReduce作业产生，现在需要告诉HBase，将这些HFile放入给定的表。这个操作将用如下命令完成：

```
HBase org.apache.hadoop.HBase.mapreduce.LoadIncrementalHFiles ch07/hfiles sensors
```

在这个命令中，我们向HBase提供了产生的HFile的位置（*ch07/hfiles*）和希望插入数据的表（sensors）。如果在文件被批量加载之前，目标表的一些region被拆分或者合并了，输入文件的拆分和合并将会在客户端被应用程序处理。事实上，用来将HFile推入HBase表的应用程序将会检验每一个HFile是否还属于单个region。如果一个region在我们推送文件之前被拆分了，加载工具会按照推入表之前同样的方式对输入文件进行拆分。另一方面，如果两个region合并了，原来分属两个region的HFile只是简单地写入相同的region即可。

当它运行时，控制台将会产生如下输出：

```
$ HBase org.apache.hadoop.HBase.mapreduce.LoadIncrementalHFiles \
                                        ch07/hfiles sensors
2015-05-18 20:09:29,701 WARN [main] mapreduce.LoadIncrementalHFiles: Skipping
non-directory hdfs://quickstart.cloudera:8020/user/cloudera/ch07/hfiles/_SUCCESS
2015-05-18 20:09:29,768 INFO [main] Configuration.deprecation: hadoop.native.lib
is deprecated. Instead, use io.native.lib.available
2015-05-18 20:09:30,476 INFO [LoadIncrementalHFiles-0] compress.CodecPool: Got
brand-new decompressor [.snappy]
```

在批量加载完成后，你应该能在HDFS的表下面找到你的文件以及它们所属的region。再看下HDFS应该会显示一些类似这样的东西（再次声明，为了适应页面宽度，文件权限和属主信息被删除了，`/HBase/data/default/sensors`被简写成`.../s`，而且region的编码名称被截取了）：

```
  0 2015-05-18 19:46 .../s/0cc...
 58 2015-05-18 19:46 .../s/0cc.../.regioninfo
  0 2015-05-18 19:46 .../s/0cc.../recovered.edits
  0 2015-05-18 19:46 .../s/0cc.../recovered.edits/2.seqid
  0 2015-05-18 20:09 .../s/0cc.../v
836 2015-05-18 19:56 .../s/0cc.../v/c0ab6873aa184cbb89c6f9d02db69e4b_SeqId_4_ ❶
```

❶ 你能看到的是，之前空着的那个region现在有一个文件。这是我们最初创建的HFile之一。通过观察文件的大小以及和MapReduce作业起初创建的HFile做比较，我们能匹配到*ch07/hfiles/v/ed40f94ee09b434ea1c55538e0632837*。你也可以看一看其他的region，并将它们和其他的输入HFile映射起来。

数据校验

既然数据已经在表中了，我们需要验证它是否和预期的一样。首先我们需要验证下是否有和预期一样多的行数。然后再验证记录的内容是否是我们所期望的数据。

表大小

使用HFilePrettyPrinter查看一个HFile文件，给出我们单个HFile内cell的数量，但是它到底代表多少唯一行数呢？由于一个HFile只代表行的一个子集，我们需要在表级别计算行数。HBase提供两个不同的机制来计算行数。

用shell计算

对于小例子而言，从shell中计算非常直接，简单和高效。它简单地做一个全表扫描并且一个接一个的计算行数。虽然这个方法对小表而言非常有效，但对于大表会花费大量的时间，因此，我们只有在确认表是小表的情况下使用这个方法。

这里是计算行数的命令：

```
hbase(main):003:0> count 'sensors', INTERVAL => 40000, CACHE => 40000
Current count: 40000, row: 9c3f
65536 row(s) in 1.1870 seconds
```

Count命令需要三个参数。第一个参数是强制的，它是你想要计算行数的表的名称。第二和第三个参数是可选的；第二个参数告诉shell每40000行显示一次进度状态，第三个参数是我们用来做全表扫描所需的高速缓存（cache）的大小。第三个参数是用来设置底层扫描对象的setCaching 值。

用MapReduce计算

第二种计算一个HBase表中行数的方法是使用MapReduce工具RowCounter。使用MapReduce来计算行数的好处是HBase将会为你表中的每个region创建一个mapper。对于一个大表而言，这将会把任务分散到多个节点来并行的执行count操作，而不是像使用shell命令计算那样，顺序地扫描每个region。

这个工具通过命令行被调用，只传入表名称即可：

```
hbase org.apache.hadoop.HBase.mapreduce.RowCounter sensors
```

这里是输出中最重要的部分（为了将注意力集中到关键信息上，同时为了减少此段的大小，某些部分被删除了）：

```
2015-05-18 20:21:02,493 INFO [main] mapreduce.Job: Counters: 31
        Map-Reduce Framework
                Map input records=65536 ❶
                Map output records=0
                Input split bytes=1304
                Spilled Records=0
                Failed Shuffles=0
```

```
                Merged Map outputs=0
                GC time elapsed (ms)=2446
                CPU time spent (ms)=48640
                Physical memory (bytes) snapshot=3187818496
                Virtual memory (bytes) snapshot=24042749952
                Total committed heap usage (bytes)=3864526848
    org.apache.hadoop.HBase.mapreduce.RowCounter$RowCounterMapper$Counters
                ROWS=65536 ❷
```

现在我们来看一看需要重要注意的部分：

❶ 由于这个作业的输入记录是HBase的行，我们将得到和HBase中行数同样多的记录数。

❷ 表中的行数将会和输入的记录数相匹配。事实上，这个MapReduce作业只是针对每一个输入记录增加了一个ROWS计数器。

HBase提供了一个叫做CellCounter的MapReduce工具，它不仅可以计算一个表中的行数，而且还能计算出表的列数以及行和列的版本数。不过，这个工具需要为表中每个唯一的行键创建一个Hadoop计数器。Hadoop默认限制计数器的个数是120个。增加这个限制是可以实现的，但是将其增加到表的行数可能引起其他问题。如果你在操作一个小的数据集，这对于测试你的应用程序并修复bug可能会有帮助。这个工具通常不能在大表上运行。关于修复这个限制的一些工作已经在Apache repository中完成。更多详细信息请参考HBASE-15773。

文件内容

我们有了自己的表、预期的行数，以及要求的格式。但是表中的数据到底是个什么样子？能否读取我们所写入的数据？可以使用以下两种方法来看看我们的数据。

使用shell

从HBase中读取数据，最简单快速的方法是使用HBase shell。使用shell，你可以发出命令来检索你想要的数据。第一个命令是`get`，它将只给你单行数据。如果你指定一个列族，只有指定的列族中的列才会返回。如果你既指定了列族又指定了列限定符（以冒号隔开），它将只会返回已存在的对应值。第二个选择是使用`scan`，它将返回一定数量的行，我们可以使用`LIMIT`参数或者`STARTROW`和`STOPROW`参数进行限制。下面的命令都将返回行键值为`000a`的行的所有列。

```
get 'sensors', '000a', {COLUMN => 'v'}
scan 'sensors', {COLUMNS => ['v'], STARTROW => '000a', LIMIT => 1 }
```

现在正如你在输出中所见，每行可能会有很多列。如果你想限制输出为一个特定的列限定符，你需要在两个命令中按照如下方式指出：

```
get 'sensors', '000a', {COLUMN => 'v:f92acb5b-079a-42bc-913a-657f270a3dc1'}
scan 'sensors', { COLUMNS => ['v:f92acb5b-079a-42bc-913a-657f270a3dc1'], \
                  STARTROW => '000a', STOPROW => '000a' }
```

那么，get命令的输出应该像这样：

```
COLUMN        CELL
 v:f9acb...   timestamp=1432088038576, value=\x08000aHf92acb5b-079a-42bc-913a...
1 row(s) in 0.0180 seconds
```

因为value是一个Avro对象，它包含有一些不可打印字符，这些字符被显示成\x08，但是大多数仍然是可读的。这表明，我们的表中含有想要的key和要找的匹配数据。

使用Java

使用shell，我们已经能够验证我们的表中包含了一些类似Avro的数据，但是为了确保它就是我们所期望看到的数据，我们需要实现一段Java代码来检索出value，把它转换成Avro对象，然后从中找出对应的部分（参见例7-4）。

例7-4：从HBase中读取Avro对象

```
try (Connection connection = ConnectionFactory.createConnection(config);
     Table sensorsTable = connection.getTable(sensorsTableName)) {❶
  Scan scan = new Scan ();
  scan.setCaching(1); ❷

  ResultScanner scanner = sensorsTable.getScanner(scan);
  Result result = scanner.next();❸
  if (result != null && !result.isEmpty()) {
    Event event = new Util().cellToEvent(result.listCells().get(0), null);❹
    LOG.info("Retrived AVRO content: " + event.toString());
  } else {
    LOG.error("Impossible to find requested cell");
  }
}
```

❶ 从HBase连接中检索表。

❷ 确保我们得到第一行之后就从扫描中返回。因为我们不想打印出多于一行的数据，不需要等待HBase返回给我们更多的数据。

❸ 对表执行扫描操作并返回结果。

❹ 将我们得到的cell作为value转换成Avro对象。

再强调一次，你可以使用Eclipse或者从命令行运行这个例子。你应该能看到类似下面显示的输出信息：

```
2015-05-20 18:30:24,214 INFO [main] ch07.ReadFromHBase: Retrieved Avro object
  with ID 000a
2015-05-20 18:30:24,215 INFO [main] ch07.ReadFromHBase: Avro content: {"id":
  "000a", "eventid": "f92acb5b-079a-42bc-913a-657f270a3dc1", "docType": "FAILURE",
  "partName": "NE-858", "partNumber": "0-0000-000", "version": 1, "payload":
  "SXOAXTPSIUFPPNUCIEVQGCIZHCEJBKGWINHKIHFRHWHNATAHAHQBFRAYLOAMQEGKLNZIFM 000a"}
```

使用这一小段代码，我们已经完成了验证过程的最后一步操作，并且从表中检索，反序列化以及打印出了Avro对象。总之，我们已经验证了HFile的文件大小，它们的格式，HFile和表中的数据条数，以及表中的内容。现在，我们可以确定，数据已经被正确地全部导入至表中了。

数据索引

实施的下一步也是最后一步就是为我们所加载的表建立索引，以便使用Solr快速的检索任意记录。建立索引是一个渐进的过程。事实上，Omneo每天接收到新的文件。和在之前章节中看到的一样，这些文件中的数据被加载进一个主表和一个索引表，其中这个主表也包含了以前的数据。我们的目标是通过将增量索引信息添加到以前所创建的Solr索引中。最终，使得该索引可检索所有已经上传到主表中的数据。为了完成最后的这个例子，你的环境中必须运行一个Solr实例。如果你习惯使用它，你可以在本地安装并运行它；不过，HBase需要以伪分布式模式运行，因为Solr的索引器不能够和localjobrunner一起工作。或者，你也可以在已经安装的虚拟机上执行这个例子。

大多数MapReduce的索引代码已经在Solr实例中被构建，被修改并简化来索引一个HBase表。

在确认你在本地已经有一个Solr环境之后，运行以下命令将会创建带有单个shard及对应schema的Slor collection。

```
export PROJECT_HOME=~/ahae/resources/ch07/search
rm -rf $PROJECT_HOME
solrctl instancedir --generate $PROJECT_HOME
mv $PROJECT_HOME/conf/schema.xml $PROJECT_HOME/conf/schema.old
cp $PROJECT_HOME/../schema.xml $PROJECT_HOME/conf/
solrctl instancedir --create Ch07-Collection $PROJECT_HOME
solrctl collection --create Ch07-Collection -s 1
```

在生产环境中，为了扩大应用，你可能要考虑使用更多的shard。

定义你的索引最重要的文件就是这个*schema.xml*文件。这个文件可以在本书的GitHub库中找到，并且含有很多的标签。这个schema最重要的部分如下：

```
<field name="id" type="string" indexed="true" stored="true" required="true"
                                                    multiValued="false" />
<field name="rowkey" type="binary" indexed="false" stored="true" omitNorms="true"
                                                     required="true"/>
<field name="eventId" type="string" indexed="true" stored="false"
                                          omitNorms="true" required="true"/>
<field name="docType" type="string" indexed="true" stored="false"
                                                    omitNorms="true"/>
<field name="partName" type="lowercase" indexed="true" stored="false"
                                                    omitNorms="true"/>
<field name="partNumber" type="lowercase" indexed="true" stored="false"
                                                    omitNorms="true"/>
<field name="version" type="long" indexed="true" stored="false" required="true"
                                                    multiValued="false" />
<field name="payload" type="string" indexed="true" stored="false" required="true"
                                                    multiValued="false" />
<field name="_version_" type="long" indexed="true" stored="true"/>
```

由于本书只关注HBase，我们将不会讲解此文件的所有部分及其细节，不过还是建议你去看看Solr的在线文档。

下面的命令将会为本例创建所需的索引：

```
export PROJECT_HOME=~/ahae/resources/ch07/search
rm -rf $PROJECT_HOME
solrctl instancedir --generate $PROJECT_HOME
mv $PROJECT_HOME/conf/schema.xml $PROJECT_HOME/conf/schema.old
cp $PROJECT_HOME/../schema.xml $PROJECT_HOME/conf/
solrctl instancedir --create Ch07-Collection $PROJECT_HOME
solrctl collection --create Ch07-Collection -s 1
```

不管基于什么原因，如果你想删除你的collection，使用如下命令：

```
solrctl collection --delete Ch07-Collection
solrctl instancedir --delete Ch07-Collection
solrctl instancedir --delete search
```

给表添加索引的步骤非常的简单。第一步，我们需要使用MapReduce扫描整个HBase表来创建Solr索引文件。第二步类似我们批量加载HFile进入HBase一样，将这些文件批量加载至Solr。限于页面大小，整个代码将不会在这里展示，不过，有一些片段还是需要在这里看一下。

例7-5 展示了需要如何在Driver类中配置我们的MapReduce作业。

例7-5：使用MapReduce driver索引HBase 的Avro表至Solr

```
scan.setCaching(500);❶
scan.setCacheBlocks(false);❷

TableMapReduceUtil.initTableMapperJob(❸
  options.inputTable,                    // 输入的HBase表名
  scan,                                  //扫描实例来控制索引
  HBaseAvroToSOLRMapper.class,           //解析单元格内容的Mapper
  Text.class,                            // Mapper的输出key
  SolrInputDocumentWritable.class,       // Mapper的输出值
  job);

FileOutputFormat.setOutputPath(job, outputReduceDir);

job.setJobName(getClass().getName() + "/"
                      + Utils.getShortClassName(HBaseAvroToSOLRMapper.class));
job.setReducerClass(SolrReducer.class);❹
job.setPartitionerClass(SolrCloudPartitioner.class);❺
job.getConfiguration().set(SolrCloudPartitioner.ZKHOST, options.zkHost);
job.getConfiguration().set(SolrCloudPartitioner.COLLECTION, options.collection);
job.getConfiguration().setInt(SolrCloudPartitioner.SHARDS, options.shards);

job.setOutputFormatClass(SolrOutputFormat.class);
SolrOutputFormat.setupSolrHomeCache(options.solrHomeDir, job);

job.setOutputKeyClass(Text.class);
job.setOutputValueClass(SolrInputDocumentWritable.class);
job.setSpeculativeExecution(false);
```

❶ 默认情况下，扫描每次只缓存一行。为了减少远程过程调用（RPC），同时提高吞吐量，我们增加了缓存的大小。

❷ 因为我们将会一次性扫描整个表，并且只扫描一次，所以将块数据缓存非但不需要，而且还会增加RegionServer中blockcache的压力。当在一个表上运行MapReduce作业的时候，较为推荐的做法是关闭blockcache。

❸ 我们再一次使用HBase实体类来配置所需要的MapReduce输入输出格式，以及所需的mapper。

❹ 使用默认的Apache Solr reducer类。

❺ 也使用默认的Apache Solr partitioner 类。

类中的一切应该都是相当容易理解的。

现在，我们一起来看看mapper。mapper的目标是读取HBase中的内容并为Solr做转换。我们已经写了一个类用于从HBase cell中创建Avro对象。在这里我们重用同样的代码，这也正是我们想要的。我们想读取每个cell，转换成一个Avro 对象，并提供给Solr我们想要索引的数据。

例7-6：使用MapReduce mapper索引HBase 的Avro表至Solr

```
event = util.cellToEvent(cell, event); ❶

inputDocument.clear();❷
inputDocument.addField("id", UUID.randomUUID().toString());❸
inputDocument.addField("rowkey", row.get());
inputDocument.addField("eventId", event.getEventId().toString());
inputDocument.addField("docType", event.getDocType().toString());
inputDocument.addField("partName", event.getPartName().toString());
inputDocument.addField("partNumber", event.getPartNumber().toString());
inputDocument.addField("version", event.getVersion());
inputDocument.addField("payload", event.getPayload().toString());

context.write(new Text(cell.getRowArray()),
              new SolrInputDocumentWritable(inputDocument)); ❹
```

❶ 为避免创建新的对象，我们重用event实例将接收到的cell转换成Avro对象。

❷ 这里再次强调，我们想要尽可能的重用现有对象，因此，只是初始化和重用了Solr的inputDocument。

❸ 将Avro event对象中我们想要索引或者存储的field分配给Solr 的inputDocument。

❹ 将Solr的document写入context用于索引。

如果你想要从命令行执行索引，你必须使用如下命令：

```
hbase -classpath ~/ahae/target/ahae.jar:`HBase classpath` \
com.architecting.ch07.MapReduceIndexerTool
```

你也可以从Eclipse中执行它而不需要指定任何参数。

数据检索

到目前为止，我们已经产生了测试数据，将数据转换成Avro对象后存储为HFile文件格式，进而将其加载到HBase表中；其次，在Solr中也创建了对应的索引，剩下的一点就是确保我们能够查询Solr得到我们的检索信息，最后基于solr的返回结果，从

HBase中检索出我们需要的信息。HBase检索的部分和之前我们看到的一样。你可以使用例7-7中的代码来查询Solr。

例7-7：基于Solr从HBase中查询Avro数据

```
CloudSolrServer solr = new CloudSolrServer("localhost:2181/solr");❶
solr.setDefaultCollection("Ch09-Collection");❷
solr.connect();

ModifiableSolrParams params = new ModifiableSolrParams();
params.set("qt", "/select");
params.set("q", "docType:ALERT AND partName:NE-555");❸

QueryResponse response = solr.query(params); ❹
SolrDocumentList docs = response.getResults();

LOG.info("Found " + docs.getNumFound() + " matching documents.");
if (docs.getNumFound() == 0) return;
byte[] firstRowKey = (byte[]) docs.get(0).getFieldValue("rowkey");
LOG.info("First document rowkey is " + Bytes.toStringBinary(firstRowKey));

//检索并打印第一个返回文档的前10列
Configuration config = HBaseConfiguration.create();
try (Connection connection = ConnectionFactory.createConnection(config);
    Admin admin = connection.getAdmin();
    Table sensorsTable = connection.getTable(sensorsTableName)) {
  Get get = new Get(firstRowKey); ❺

  Result result = sensorsTable.get(get);
  Event event = null;
  if (result != null && !result.isEmpty()) {❻
    for (int index = 0; index < 10; index++) { //打印第一个10列
      if (!result.advance())
        break; // There are no more columns and we have not reached 10
      event = new Util().cellToEvent(result.current(), event);
      LOG.info("Retrieved AVRO content: " + event.toString());
    }
  } else {
    LOG.error("Impossible to find requested cell");
  }
}
```

❶ 连接到你的Solr集群。如果你的Solr和HBase不在同一个集群中运行，这里需要调整下。

❷ 定义你想使用的Solr collection。

❸ 配置你想让Solr执行的请求。这里我们请求所有ALERT文档的NE-555部分。

❹ 执行Solr请求并检索来自服务器的响应。

❺ 调用HBase，指定由Solr返回的第一个document的行键。

❻ 遍历给定key的所有列并且展示从这些列中检索出的前10个Avro对象。

更进一步

如果你想扩展本章展示的例子，下面的列表是基于我们本章讨论的内容提出的一些可供你尝试的选择。

输入文件更大一些

为了确保例子能够快速运行，我们使用的数据集非常的小。如果使用一个更大的数据集会怎么样？根据你可用的硬盘空间和环境的性能，尝试创建一个非常大的输入文件并用同样的方式检验它的处理过程。

单个region的表

我们创建了一个基于我们使用的key切分成多个region的表，因为这是避免热点的好方法。不同的MapReduce作业也因此产生了多个文件，每个region产生一个。如果我们创建一个只有单个region的表会怎么样？尝试修改建表语句，使其只有一个region并加载超过10GB的数据进去。你应该能在数据插入之后看到region splitting；不过，由于我们使用的批量加载，你在此region上仍然看不到任何热点问题。你可以查看HDFS来验证你的表切分及每个region的内容，正如我们在本章前面“批量加载”中讨论的一样。

表参数的影响

我们所创建的表使用的参数适用我们当前的用例。我们建议修改各种参数并重新运行此过程来测试各参数的影响。

压缩

尝试使用不同类型的压缩并进行比较。如果你使用Snappy（它很快），那就试试配置成LZ4（它可能慢一些，但是压缩的更好），并且比较下处理与文件大小相关的所有一切总共花费的时间。

块编码

由于我们存储进此表的key的格式，我们配置使用FAST_DIFF来对块数据进行编码。这里再说一下，看一看效率以及最终整体数据的大小。

布隆过滤器

当做读取操作的时候，布隆过滤器非常有用，它会跳过那些我们要找的key并不存在的HBase存储文件。不过，在这里我们知道，我们要找的数据在文件中都是存在的，因此禁用了布隆过滤器。创建一个列表，包含有几十个表中存在

的行和列，测试一下读取整个列表需要多长时间。激活表上的布隆过滤器并运行一个大合并，将会触发布隆过滤。针对这个用例而言，后续的测试应该不会有明显的性能提升。

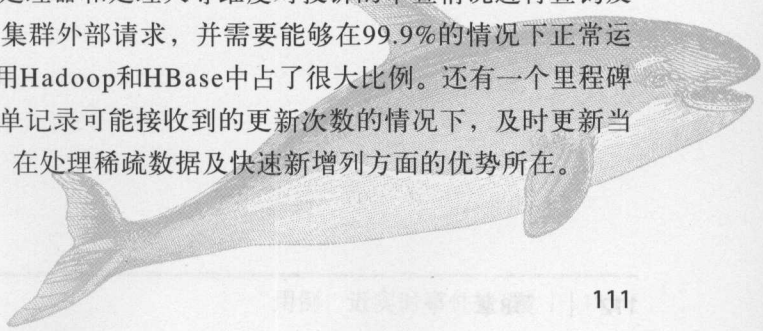
激活布隆过滤器通常情况下是很好的。在这里我们禁用了它，是因为这个用例很特殊。如果你不确定的话，就将它开启吧。

第8章

用例：近实时事件处理

下面一个用例主要涉及医疗行业中的索赔问题。索赔处理器利用软件即服务（SaaS）模式作为病人、医院和医生办公室之间的一个桥梁。因为对于许多医院和医疗保健提供者而言，医疗索赔很难管理，索赔处理器（也称为交互处理中心）接受来自客户的索赔（如医院和医生办公室），然后对这些投诉做些标准流程化处理后，再发送给处理人。这是一个既困难又乏味的过程，因为大多数医院和医生办公室没有使用标准流程化处理投诉的习惯。这是信息交换中心专门为客户设计的统一格式并提交所有索赔的地方。

与以前的用例相比，交互中心需要一些额外复杂层。例如，这个数据不仅需要通过机器去生成，并被批量加载，而且传入的数据也需要快速被处理。此外，由于数据分为个人可识别信息（PII）或受保护健康信息（PHI），因此会受到健康保险流通与责任法案（HIPAA）的保护，它需要大量的安全层级控制。这个案例专注于电子医疗索赔记录以及创建一个系统来处理这些记录。该系统的最终目标是降低典型的投诉索赔时间，从30天最终做到实时的投诉处理，通过优化索赔，以确保能呈现并覆盖所有必要信息，并可在将索赔信息发送给保险提供商之前，迅速进行诉求驳回。在类似这样的系统中，理想的服务级别协议（SLA）是能够做到在不到15秒的时间内，就可以从客户、投诉处理器和处理人等维度对投诉的审查情况进行查询及详阅。由于系统需要实时服务集群外部请求，并需要能够在99.9%的情况下正常运行，这在直接决定是否需要使用Hadoop和HBase中占了很大比例。还有一个里程碑的概念，其需要能够在不知道单记录可能接收到的更新次数的情况下，及时更新当前的投诉记录。这正是HBase，在处理稀疏数据及快速新增列方面的优势所在。



索赔提供商尝试采用许多其他的系统来解决这个用例。当评估这个项目的技术时，提供商很幸运，因为这是一个特别新的项目。这是一个可供选择的技术的白板。首先看的是Oracle RAC技术，这是当时现有的数据库，运营商在预定的数据体量上进行了一些负载测试，但不幸的是，RAC无法跟上。接下来是Netezza公司，由于在小文件的处理上以及总开销成本不足而被否决了。最后对Cassandra进行了评估。因为该用例需要访问不同的处理层，它也被排除在完整的Hadoop生态系统支持之外。目前集群在HDFS的实例上有大约178 TB的数据，在HBase实例上大约有40TB的数据，为每秒处理大约30 000个事件，并且每秒从Solr接收大约10个请求。

如前所述，此群集当前包含PHI数据。存储此类受保护数据的总体要求不仅限于技术，也受限于处理过程。我们主要关注在系统中注存储这些数据的技术要求，但这些严格的准则也在很大程度上决定了技术的选型。这些准则都需要被很严肃认真的对待，因为如果发现系统的管理人员违反法律，他们可能遭受大额罚款和监禁。当处理PHI的时候，投诉运营商需要提供一个强大的身份验证，授权和加密处理。

该投诉处理运营商希望在部署系统之前确保满足所有级别的要求。对于认证层，运营商和Kerberos的行业标准保持一致。可以从几个不同的地方控制授权；一是利用内置于HBase的ACL，二是通过HDFS的ACL，最后是通过搜索及Impala层进行认证控制。加密层刚开始采用了Vormetric加密，这似乎并没有造成一个大的性能冲击，但后来的运营商选择了Gazzang加密，利用集成密钥库和AES-NI来降低CPU的负载。同时运营商也利用Cloudera Navigator来做审计，数据谱系和元数据标签。

图8-1展示了整体架构。起初，这个图可能有点令人望而生畏，因为它包含了Hadoop生态系统的一些技术，而我们却没有回顾到。在深入到不同的层之前，让我们花一分钟稍微深入了解一下Kafka和Storm，它们在近实时管道处理中发挥了重要作用。

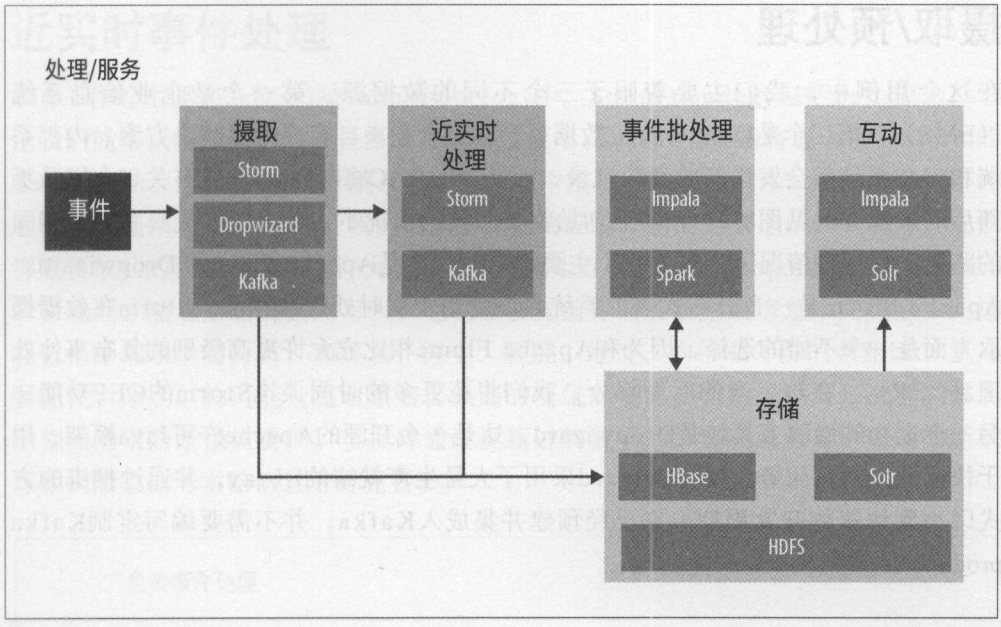


图8-1：体系结构概述

Kafka在实时处理领域是个常用的组件。Kafka维持的不同消息流被称为topic。发布消息到Kafka topic的进程被称为producer。可以预先创建很多producer（类似Flume的source），或你可以根据用例的要求，总是写你自己的producer。从topic获得数据的进程被称为consumer。consumer也通常负责写入到下游应用。Kafka通常是由一个或多个节点组成的集群，每个节点被称为broker。这些consumer分成consumer group，每一个group处理自己的偏移，使consumer能够轻松订阅和退订topic，同时不影响其他consumer从topic中读取数据。Kafka可以利用Kafka节点内的众多spindle来调整分区，从而实现非常高的读写吞吐量。

既然在实际中你已经对新技术有所理解，我们将架构分成三个不同的类别：

- 摄取/预处理。
- 近实时事件处理。
- 处理/服务。

摄取/预处理

在这个用例中，我们主要着眼于三个不同的数据源。第一个是企业信息系统（EMS），第二个是Oracle RAC数据库，第三个是来自客户内部解决方案。内部系统和EMS系统都会发送新的索赔记录，而Oracle RAC将向kafka发送有关每个记录更新后的里程碑（见图8-2）。缺乏初始收集工具的情况下，每个摄取流具有大致相同的路径。在这种情况下，需要两个主要工具：首先是Apache Storm和Dropwizard。Apache Storm是一个分布式摄取系统，设计用于实时处理数据流。Storm在数据摄取方面是一个不错的选择，因为和Apache Flume相比它允许更高级别的复杂事件处理（CEP）。在这一章的后面部分，我们将花更多的时间谈论Storm的CEP功能。另一个最初的摄取工具就是Dropwizard，这是一个开源的Apache许可Java框架，用于快速开发网页服务。Dropwizard采用了大量生产就绪的libray，并通过捆绑的方式以达到快速的开发周期。它已经预建并集成入Kafka，并不需要编写定制Kafka producer。

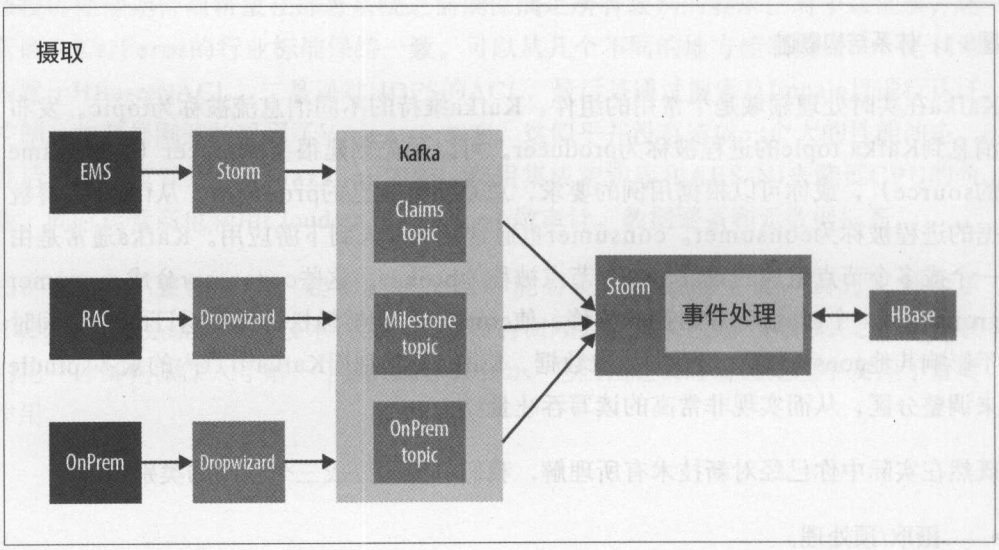


图8-2：详细的摄取管道

在该用例中，我们使用了三个独立的topic，每个topic用于每个摄取路径。这允许终端用户能够添加其他系统以从单个topic读取，而不必接收其他不必要的数据。在这种情况下，众多的topic也被简单的使用以方便管理和未来的扩展。我们有一个单独的Storm topology，从Kafka的三个主要的topic中拉数据。

近实时事件处理

ETL工作的主要部分位于该Storm的topology中。Storm topology包括Bolts概念，它让我们能够执行所需的CEP任务。在这种情况下，kafka消费端将成为用于Storm的Kafka spout。这时Storm bolt将从kafka spout中接收各种类型事件，此处将会对事件进行完善操作（见图8-3）。我们有两个流需要处理：新的索赔记录和里程碑。它们都需要单独的处理，其中允许大量的数据流发生在一个单一的topology，使得Storm容易处理。随着索赔数据的写入，它们需要先前存储的数据或信息来完善它。此过程包括添加缺少的特质，如姓名、医生、医院、地址、内部和外部的索赔ID。重点要记住，命名结构可能会随着索赔提供商的变化而变化。索赔提供商每次都能够识别相同的索赔是很重要的。里程碑通过更新过的数据和那些在服务层中使用的新状态级别的数据对索赔信息进行完善。

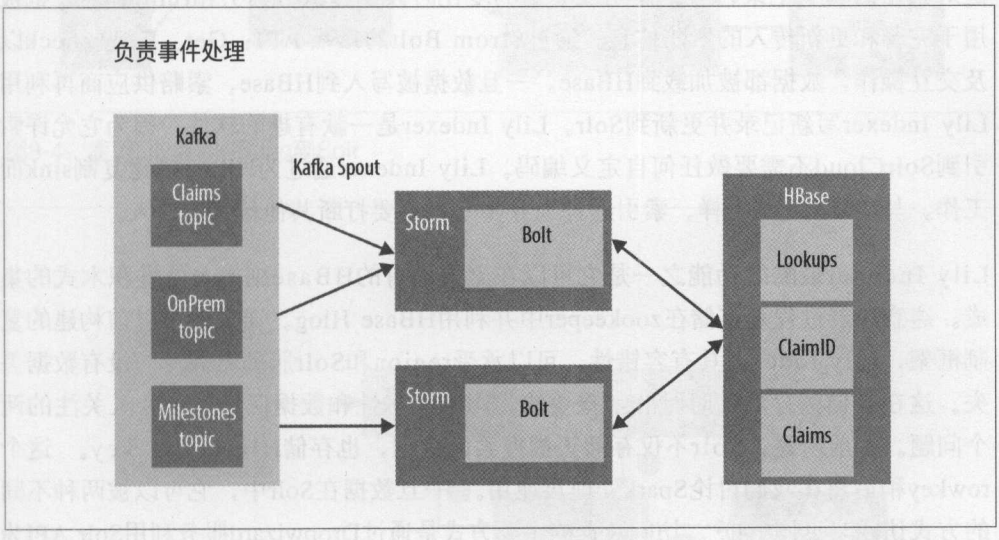


图8-3：飞行中的事件处理

HBase的rowkey的设计必须仔细考虑PII数据的处理。处理医疗信息的最重要的要求之一是保证永远不会暴露错误的客户信息的能力。在这种情况下，该rowkey是客户ID和随机哈希的关联。下面是三张相关表：

- Lookups - InternalID
- ClaimsID - claimsID
- Claims - CustomerID|random hash

客户ID必须是rowkey的第一部分，因为这保证当应用程序在HBase中查找时，只返回该记录。问题是大多数客户ID倾向于在系统中增加，并且通常只是一系列的数字。通过添加随机散列到密钥的结尾，可以达到额外的字典排序分的目的。最理想的row key将是一个简单的具有很低的碰撞率的MD5哈希，这会将使得其非常均匀的分布，写入路径也更平滑。然而，当处理PII数据时，不管风险有多低，使用MD5哈希将是一场赌博。记住，人们每天都会赢得彩票！

处理/服务

我们已经看了管道流的摄取和CEP层。我们的一切讨论发生在集群外部，除了storm bolt调用HBase中的get和put。索赔提供商结合使用Impala、Spark、Solr和全能的HBase（这是本书的重点）的组合。在这种情况下，HBase以多种方式被使用：除了是对索赔的真实记录，它也被用在数据的变化时候重建Solr索引和Impala表，也被用于完善和更新传入的索赔信息。通过Strom Bolt的Java API：Get、Put、check以及交互操作，数据都被加载到HBase。一旦数据被写入到HBase，索赔供应商再利用Lily Indexer写新记录并更新到Solr。Lily Indexer是一款有趣的软件，因为它允许索引到SolrCloud不需要做任何自定义编码。Lily Indexer通过为HBase创建复制sink而工作。与HBase复制一样，索引过程为异步写，不要打断其他传入的写入。

Lily Indexer最酷的功能之一是它可以在之前已有的HBase副本上做堆积木式的集成。这意味着过程是存储在zookeeper中并利用HBase Hlog。通过使用以前构建的复制框架，Lily Indexer具有容错性，可以承受region和Solr服务器故障而没有数据丢失。这在存储医疗索赔时时非常重要的，因为准确性和数据保护是最令人关注的两个问题。如前所述，Solr不仅存储正被搜索的数据，也存储HBase row key。这个rowkey稍后将在我们讨论Spark处理层使用。一旦数据在Solr中，它可以被两种不同的方式访问（见图8-4）。访问数据的主要方式是通过Dropwizard服务利用Solr API为终端用户提取最新的索赔数据和关键的信息数据。另一个方式是借助Spark，在上述数据集上运行特殊的机器学习算法。

Spark正被用于识别缺失关键部分数据以及处理字段中出现的不正确数据的索赔场景。Solr可以完美的处理这些问题，并可以作为二级、三级、四级等索引，因为HBase默认只能查询rowkey。我们做的第一件事是查询Solr来获得我们想要的特定记录；这些记录也将包含HBase rowkey。一旦我们有了HBase rowkey，基于从Solr查询的结果，我们就可以执行get/ multiget和scan。一旦我们从HBase获得数据，一系列的数据模型（抱歉，不想模糊，但我们确实需要保护一些知识产权）被应用于结

果集。到时候，将Spark的输出发送到告警系统，这将触发里程碑以提醒客户，有丢失的字段，或其他需要填写完相应的属性值才能继续后续流程（见图8-5）。

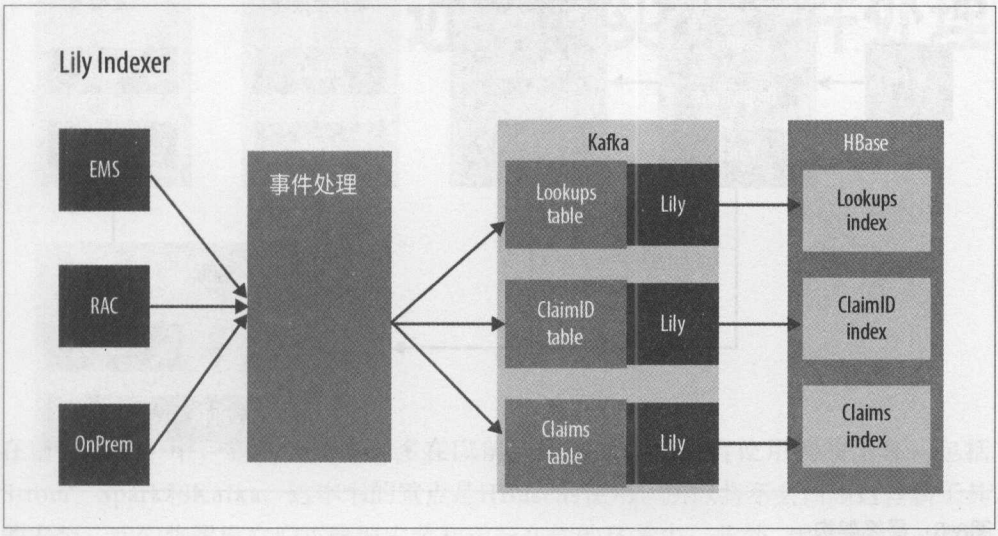


图8-4：使用Lily复制HBase到Solr

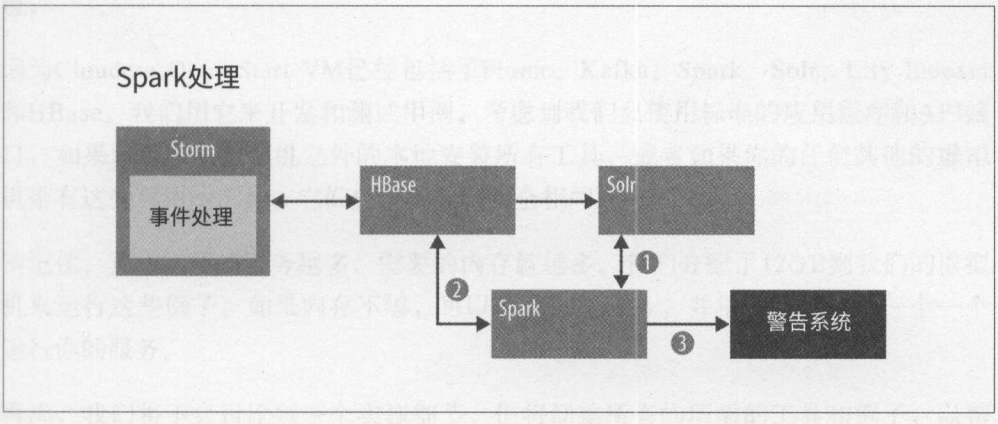


图8-5：Spark处理流程

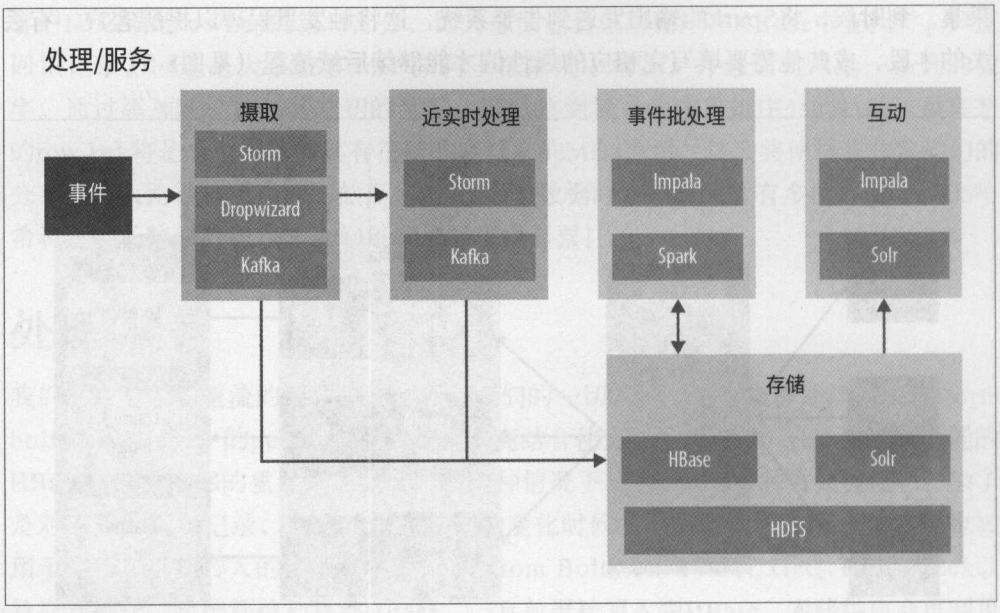

处理/服务
事件
摄取
Storm
Dropwizard
Kafka
近实时处理
Storm
Kafka
事件批处理
Impala
Spark
互动
Impala
Solr
存储
HBase
Solr
HDFS

图8-6：最终架构

第9章

近实时实现事件处理

在前面的章节中，我们介绍了许多在以前实施的用例中没有使用到的工具，包括Strom、Spark和Kafka。这本书的重点是HBase的使用，所以将不会回顾这些新工具的安装，我们将假设它们正确地安装和运行在你的环境中。此外，因为Twitter发现了Storm设计和可扩展性方面的缺陷并淘汰了它[注1]，这里的例子我们使用Flume来实现。

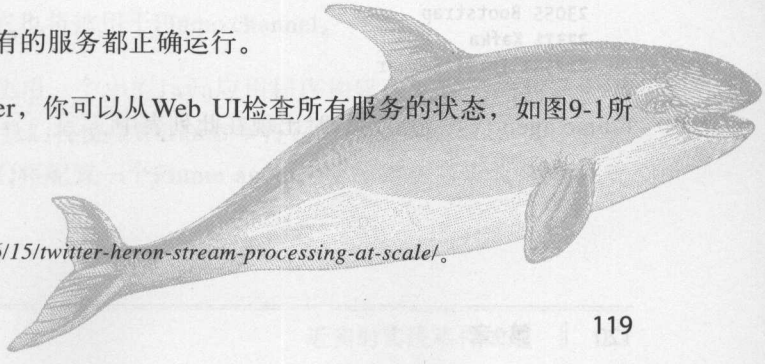
因为Cloudera QuickStart VM已经包括了Flume、Kafka、Spark、Solr、Lily Indexer和HBase。我们用它来开发和测试用例。考虑到我们只使用标准的应用程序和API接口，如果你可以在虚拟机之外的本地安装所有工具，或者如果你的任何其他的虚拟机带有这些可用的工具，它们应该都会以完全相同的方式工作。

请记住，正在运行的服务越多，需要的内存就越多。我们分配了12GB到我们的虚拟机来运行这些例子。如果内存不够，可以停止这些服务，并根据运行步骤一个一个运行你的服务。

再次，我们将不会讨论每一个实施细节，但将涵盖所有的所需的工具和例子，以帮助你了解在此阶段中重要的内容。

在我们开始之前，需要确保所有的服务都正确运行。

如果使用的是Cloudera Manager，你可以从Web UI检查所有服务的状态，如图9-1所示。



注1：*http://blog.aclyer.org/2015/06/15/twitter-heron-stream-processing-at-scale/*。

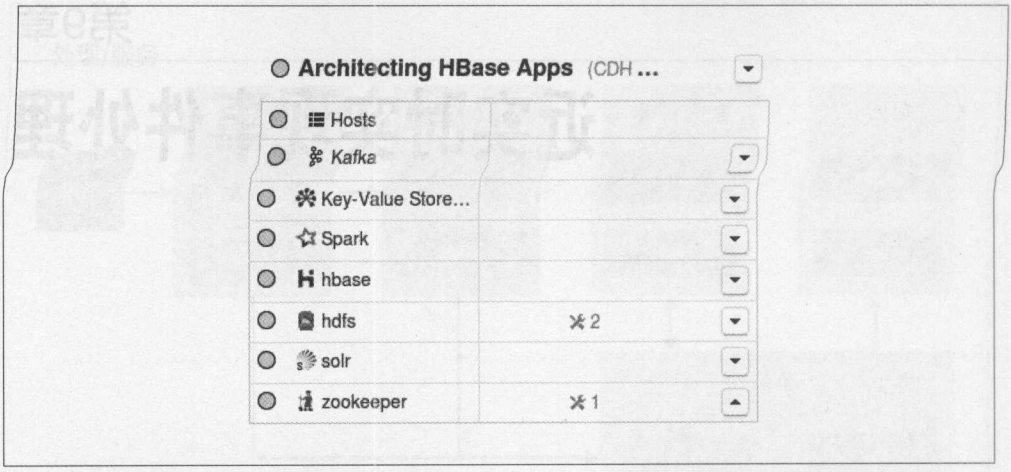


图9-1：Cloudera Manager中的集群服务

如果你没有使用Cloudera Manager，或者如果你没有任何Web UI，可以使用sudo JPS命令来确保所有服务正在运行。

重要的服务在这里加粗显示：

```
$ sudo jps
12867 Main
5230 Main
12735 EventCatcherService
22423 DataNode
12794 Main
22920 HMaster
12813 HeadlampServer
12753 NavServer
12841 AlertPublisher
22462 SecondaryNameNode
22401 NameNode
22899 HRegionServer
22224 QuorumPeerMain
29753 Jps
12891 NavigatorMain
24098 Application
24064 Main
23055 Bootstrap
22371 Kafka
822962 ThriftServer
```

Flume agent仅在它将运行时出现在此列表中。

应用流

如在第8章中所述，Flume将从外部数据源中提取数据并存储到Kafka中以便排队。然后另一个Flume agent会读取不同的Kafka队列，如果需要的话，将会对数据进行处理，然后将它发送到HBase进行存储，在HBase中Lily Indexer将获取数据并发送给Solr来构建索引。这是我们将要在这里实现的。我们不会实现所有的source以及所有的实时处理选项，但我们将确保从最初的Flume事件直至Solr索引之间有一个完整的数据流。为了使得摄取测试事件进入我们的流程变得简单，我们将使用Flume直接从Kafka队列读取事件，这样，我们需通过`kafka-console-producer`进行测试数据的插入。

图9-2展示了数据流的一个简化版本。

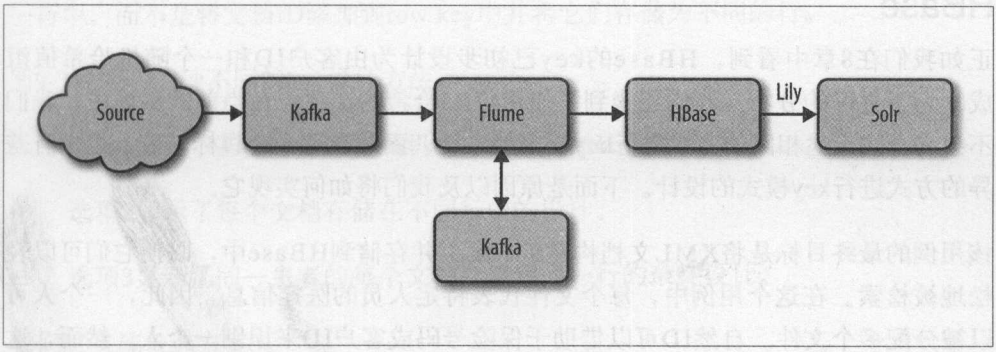


图9-2：数据流

为了达到这个示例的目的，假设传入的数据为XML文件，我们将把这个文件转换成Avro对象，然后将其插入HBase并把所有的字段分别构建为Solr索引。

Kafka

因为Kafka将是所有数据的切入点，我们需要做的第一件事就是运行和配置Kafka。Kafka将被用于两个目的。首先，它将从外部源接收到的所有数据存储到队列中，并使其可用于下游用户。其次，它也将被用于Flume channel。

为了填充Kafka队列，我们将使用一个小型Java应用程序构建我们自己的数据生成器，以创建随机的XML文档。我们将使用Kafka命令将它们加载到Kafka。为了能够读取和处理所有这些事件，我们将配置一个Flume agent，使用拦截器来执行XML到Avro转换。

Flume

Flume是一种流数据处理引擎，经常用于从source引入数据到sink，如果需要则可以应用于非常小的修改。另外，它还需要配置一个channel。在channel中，Flume可存储传入的数据直到它被发送到目的地。

在最初的项目实现中，我们选择了Storm，但是，我们感觉到用Flume更好，这也是社区广泛采用和使用的。

在此类的实例中，我们将使用Flume来读取Kafka source的事件，并进行相应的转换与完善，然后将它们推到HBase。在这之中，我们需要使用一个Kafka source，一个Kafka channel、拦截器，以及一个HBase sink。

HBase

正如我们在8章中看到，HBase的key已初步设计为由客户ID和一个随机哈希值组成。为了更好的分布，我们还谈到了使用MD5哈希的选项。在当前的实施中，我们不会完全按上述相同的方式进行key模式的设计，而是通过一个目标一致，但稍有差异的方式进行key模式的设计。下面是原因以及我们将如何实现它。

该用例的最终目标是将XML文档构建Solr索引并存储到HBase中，以便它们可以轻松地被检索。在这个用例中，每个文件代表特定人员的医疗信息。因此，一个人可以被分配多个文件。自然ID可以借助于保险号码或客户ID来识别一个人。然而，这个ID的分布不能得到保证，可能会导致一些region负载不均。由于这个原因（通过保险号码做scan是不可靠的），我们想尝试对这个数字进行哈希处理。正如我们在之前的用例中看到的那样，对key做哈希，可以让它有一个更好的分布。所以我们的key可以是保险号码的MD5值。即使MD5值的碰撞风险非常低，但风险仍然存在。如果在检索一个病人的医疗记录时，由于一个MD5的碰撞，我们也检索到另一个病人的记录。这可能造成混乱，可能导致错误的诊断，并可能造成非常严重的后果，包括法律后果。数据冲突在医学界是根本不可接受的。这意味着我们需要找到一种方法来保证MD5的分布来确定永远不会存在任何碰撞。实现这一目标的最简单的方法是在MD5末尾简单地添加客户ID或保险ID。因此，即使两个不同的ID产生同样的MD5值，它们仍然可以被用来作为一个与众不同的key，在HBase中，每个人都会有属于自己的row key。对于返回行，需要ID和MD5组成的值与key完全匹配，这使得它是唯一的。与之前的章节中提及的key相比，此设置能够更好的将数据分布在数据表中，但是以更大的存储空间作为代价的。实际上，MD5加上ID所占的空间将大于

ID和一些随机字节。然而，这种额外的代价在改善表的分布和简化表region分裂方面被证明是值得的。

正如我们已经指出的，做哈希的目的是改进整个表key的分布。MD5的哈希值为16字节。但要实现良好的分布，只需几个字节就足够了。因此，我们将只保留前两个字节，并以字符串格式存储。 我们选择使用MD5哈希，因为已经在第8章的例子中使用到它了，但任何能够提供均匀分布的其他种类的哈希值也可以用于替换MD5哈希（例如，如果你愿意，也可以使用CRC32）。最后，因为我们必须对rowkey做索引，它将更容易存储为可打印字符串而不是字节数组。前四个字符将表示哈希值，而key的剩余字符将表示保险号。

此外，即使对于正在接受严重疾病治疗的患者，我们也不会期望每人拥有数百万份文档，实际上限制为10 000份文档似乎合理。 这允许我们将所有这些文档存储在同一行中，而不是将文档ID添加到row key中并将它们存储为不同的行。

图9-3说明了三种不同的key设计方法：

- 选项1显示了基于客户ID和随机哈希值的初始key设计。
- 选项2显示了每个文档存储在不同行中的设计。
- 选项3显示了同一患者的每个文档存储到同一行的最终设计。

我们将实施备选方案3。

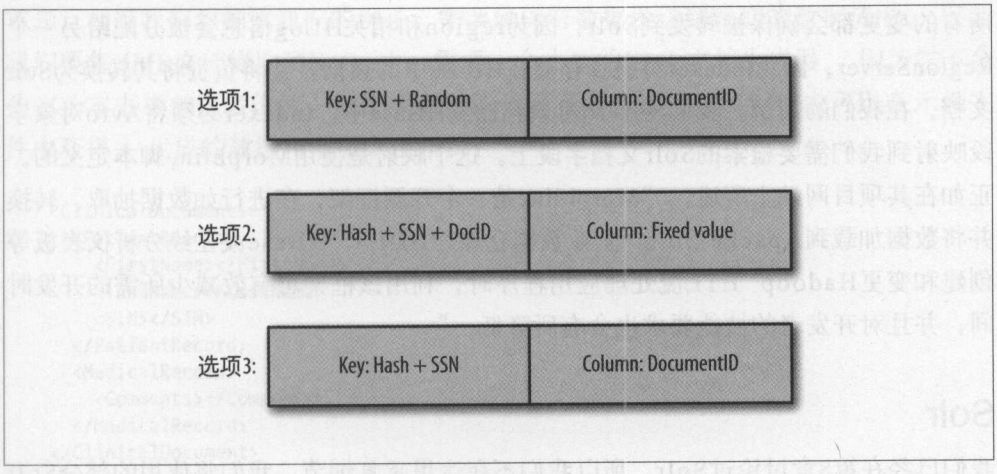


图9-3：Key的选项

你可能会想知道为什么我们决定使用选项3而不是选项2。在最后两个选项中，文档ID一次存储在key中，一次作为列限定符。因为不管任何选项被选中，我们在尾部都会存储相同的信息，两个选项的存储大小将是相同的。再次，在这两种情况下，当检索文档时，我们将查询HBase给定的客户ID及其MD5和文档ID。所以对于这两种情况，访问模式将是直接和一致的。

其中的主要区别是将文档存储在一起可以让你从HBase行级一致性中受益。事实上，如果你的上游系统想同时上传两个文件，并保证它们都上传成功或者失败，让它们在同一行，将允许你实现这一目标。然而，将它们放在两个不同的行上可能会将文档归入两个不同的region。这在部分故障的情况下是有问题的。如果你没有任何一致性约束，为了提高可扩展性，使用版本2中描述的设计是完全合适的，甚至是首选的。

最后，请记住HBase永远不会在一行内进行分裂。所以这种设计方式只有当你的行数为一个合适并有限的大小才会起作用。这就是为什么我们估计一行可能实际具有的最大列数。如果我们认为每行记录最多有10 000列，并且每列都是小于10KB的Avro对象，那此特定行大约为100 MB数据。假定HBase region可以轻松超过10 GB，这为我们提供了足够的空间来存储这些异常行。

Lily

Lily Indexer的目标是将HBase中接收到的所有突变复制到Solr中。我们不会分析所有的技术细节，但Lily是建立在HBase副本框架之上。因此，即使发生了节点故障，所有的变更都会确保被转发到Solr，因为region和相关HLog信息会被分配给另一个RegionServer。因为Indexer将接收存储在HBase中的数据，并将负责将其转换为Solr文档。在我们的用例，我们将Avro对象存储到HBase中。Indexer必须将Avro对象字段映射到我们需要检索的Solr文档字段上。这个映射是使用Morphline脚本定义的。正如在其项目网站上所述，“Morphline是一个开源框架，在进行如数据抽取、转换并将数据加载到Apache Solr，企业数据仓库，HDFS，HBase或在线分析仪表板等创建和变更Hadoop ETL流处理应用程序时，利用该框架可有效减少所需的开发时间，并且对开发者的技能要求也会有所降低。”

Solr

我们已经在第8章讨论过Solr，所以我们不在这里重复细节。我们将使用的部分Solr schema如下：

```
<field name="id" type="string" indexed="true" stored="true" required="true"
                                                    multiValued="false" />
<field name="rowkey" type="binary" indexed="false" stored="true" omitNorms="true"
                                                        required="true"/>
<field name="documentId" type="string" indexed="true" stored="true"\
                                        omitNorms="true" required="true"/>
<field name="payload" type="text_general" indexed="true" stored="false"
                                      required="true" multiValued="false" />
<field name="_version_" type="long" indexed="true" stored="true"/>
```

再次，完整的schema可参考本书的示例文件。

实施

现在我们已经定义了所有的主要组件（Kafka、Flume、HBase、Lily和Solr），接下来是确保数据可以从入口点流向Solr索引的时候了。为了让你在进入本章之前能够测试这些例子，我们会按照与数据流相同的顺序实现不同要求的部分。

在进入所有细节之前，我们建议直接从创建HBase表开始。事实上，在表已经创建后，才允许你测试我们所描述的例子。稍后，我们将回顾此表创建的详细信息并提供更高效的创建脚本，但为了要快速启动，只需在HBase shell中输入以下内容即可：

```
create 'documents', 'c'
```

数据生成

首先，如果我们想测试我们的例子，需要确保有数据发送到我们的数据流中。 因为我们要将XML文档提取到Kafka中，需要一个小型的XML文档生成器。 因为它不会为它为本书增加很多价值，这个生成器的代码不会在这里打印，但它可以在示例文件中获得。 生成的数据将遵循以下格式：

```
<ClinicalDocument>
  <PatientRecord>
    <FirstName></FirstName>
    <LastName></LastName>
    <SIN></SIN>
  </PatientRecord>
  <MedicalRecord>
    <Comments></Comments>
  </MedicalRecord>
</ClinicalDocument>
```

当然，现实世界的医学文档所包含的字段比我们在这里生成的字段多的多。然而，我们在这里使用的几个字段足以实现例子。社会保险号是一个必填字段，但其他字段可能并不总是被填充。在摄取过程中，当字段丢失时，HBase将尝试查找缺少的信息。

为了生成这些文档，我们需要做的第一件事是创建一个随机的社会保险号码（SIN）。 我们希望同样的SIN总是代表同一个人。因此，我们将根据这个数字生成一个名字和一个姓氏。 这样，相同的号码将总是返回相同的姓名。但是，因为我们也想要演示拦截器的使用，我们要生成一些没有名字或姓氏的消息。 利用Flume拦截器，我们可以添加字段缺失检测。 这不仅会转换来自XML的消息到Avro，也将允许HBase执行Get命令以检索之前可能包含缺少字段的消息。这将允许我们将文件补充完整。

要运行XML文档生成器，在命令行中，只需使用以下命令：

```
java -classpath ~/ahae/target/ahae.jar com.architecting.ch09.XMLGenerator
```

Kafka

现在，我们已做好将消息推送入数据流的准备，接下来，我们需要准备Kafka去接收它们。

确保Kafka配置了zookeeper.chroot属性来指向/kafka zookeeper路径。 事实上，默认情况下，一些发行版会将Kafka文件夹保留为ZooKeeper根路径。如果使用默认值，最后，你将在根路径下创建所有的Kafka文件夹，区分这些Kafka文件夹和其他应用的文件夹会变得令人困惑。 此外，不设置值可能导致出现以下错误：Path length must be > 0。

第一步是创建一个Kafka队列，实现方式如下：

```
[cloudera@quickstart ~]$ kafka-topics --create --topic documents --partitions 1 \
                         --zookeeper localhost/kafka --replication-factor 1
Created topic "documents".
```

在生产过程中，你很可能想要更多的分区和较大的复制因子。但是，在一个只有单个Kafka服务器运行的本地环境中，你将无法使用任何更大的参数。

要了解此命令可接受的不同参数的详细信息，请参阅Kafka在线文档。

创建队列后，有多种方法可以向其中添加消息。其中最有效的方法是使用Kafka Java API。但是，为了让示例简单明了，接下来我们将使用命令行API。要将消息添加到新生成的Kafka队列中，请使用以下命令：

```
java -classpath ~/ahae/target/ahae.jar com.architecting.ch09.XMLGenerator | \
kafka-console-producer --topic documents --broker-list localhost:9092
```

此方法将调用我们之前实现的XML生成器，并将使用它的输出作为`kafka-console-producer`的输入。在这个调用结束时，一个新的XML消息将被创建并推送到Kafka队列中。要验证你的topic是否包含生成的消息，可以使用以下命令：

```
kafka-console-consumer --zookeeper localhost/kafka --topic documents \
                       --from-beginning
```

此命令将连接到`documents` topic，并从第一个可用的开始，输出本topic中所有的事件。

最后一个命令的输出应该如下所示：

```
<ClinicalDocument>\n<PatientRecord>\n<FirstName>Tom</FirstName>\n<LastName>...
```

这将表示至少有一个事件可用于Kafka topic。

此外，因为在我们配置Flume时，使用Kafka作为channel，我们还需要使用以下命令创建Flume Kafka channel：

```
[cloudera@quickstart ~]$ kafka-topics --create --topic flumechannel \
   --partitions 1 --zookeeper localhost/kafka --replication-factor 1
Created topic "flumechannel".
```

Flume

现在我们的topic已经获得事件，接下来需要消费它们并将数据存储到HBase中。

Flume的配置是通过一个属性文件完成的，我们可以在该文件中定义所有的配置属性。Flume参数的格式如下：

```
<agent-name>.<component-type>.<component-name>.<parameter> = <value>
```

我们将使用摄取作为所有配置的代理名称。

有关Flume的工作原理及其所有参数的详细信息，请查看项目的在线文档或Hari Shreedharan的著作《Using Flume》（O'Reilly）。对于Kafka集成Flume的所有详尽参数，请参阅发布于Cloudera Engineering上的博客文章："Flafka:Apache Flume Meets Apache Kafka for Event Processing"。

Flume Kafka source

Flume可以配置许多不同的source，其中大多数已经开发好了。如果你正在寻找的source没有建立，你可以自己开发。我们这边使用了Kafka source。我们将使用以下参数通知Flume有关此source的信息：

```
ingest.sources = ingestKafkaSource
```

这就告诉Flume，ingest代理只有一个名为ingestKafka的source。现在我们已经告诉Flume我们有一个source，接下来我们必须配置它：

```
ingest.sources.ingestKafkaSource.type = \
    org.apache.flume.source.kafka.KafkaSource
ingest.sources.ingestKafkaSource.zookeeperConnect = localhost:2181/kafka
ingest.sources.ingestKafkaSource.topic = documents
ingest.sources.ingestKafkaSource.batchSize = 10
ingest.sources.ingestKafkaSource.channels = ingestKafkaChannel
```

同样，这为Flume提供了我们定义的源的所有细节。

Flume Kafka channel

Flume channel是Flume中用作source和sink之间的缓冲区。Flume将使用此channel存储那些从source读取并等待被发送到接收器的事件。 Flume支持几种不同的channel选项。memory channel是非常有效的，但是在服务器故障的情况下，存储到这个channel的数据是会丢失的。此外，即使服务器的内存空间越来越大，但和磁盘的存储容量相比，它们仍然有限的。Disk channel允许数据在服务器故障的情况下持久化数据。但是，这将比其他channel慢，并且在用于存储channel的磁盘不能被恢复的情况下仍然会存在数据丢失。与其他channel相比，使用Kafka channel将占用更多的网络资源，但数据可以持久化在Kafka集群中，因此不会丢失。在传送数据到Flume之前Kafka主要将信息存储在内存中，这将减少磁盘的延迟对应用程序的影响。此外，如果source提供事件的速度比sink处理的速度更快，则Kafka集群可以比单个disck channel更具优势，并且将允许存储更多待处理的数据，以便稍后source放缓之后处理。在我们的用例中，我们存储医疗信息，不能丢失任何数据。因此，我们将使用Kafka队列作为Flume channel。

channel配置与source配置类似：

```
ingest.channels = ingestKafkaChannel
ingest.channels.ingestKafkaChannel.type = org.a.f.channel.kafka
ingest.channels.ingestKafkaChannel.brokerList = localhost:9092
ingest.channels.ingestKafkaChannel.topic = flumechannel
ingest.channels.ingestKafkaChannel.zookeeperConnect = localhost:2181/kafka
```

这告诉我们，摄取代理使用一个名为ingestKafkaChannel的kafka channel。

Flume HBase sink

Flume sink的目标是从channel中获取事件并将它们存储在下游。在这里，我们将HBase视为存储平台。因此，我们会配置一个HBase Flume sink。Flume带有几个默认的序列化器，可将数据推送到HBase。序列化的目标是将Flume事件转换为HBase事件（除此之外没有别的）。然而，即使它们可以在大多数情况下可以使用，但那些序列器并不能完全控制行键和列名。我们将不得不实现我们自己的序列化程序，以便能够从Avro对象中提取我们的行键。可以使用以下操作来完成sink配置：

```
ingest.sinks = ingestHBaseSink
ingest.sinks.ingestHBaseSink.type = HBase
ingest.sinks.ingestHBaseSink.table = documents
ingest.sinks.ingestHBaseSink.columnFamily = c
ingest.sinks.ingestHBaseSink.serializer = \
    com.architecting.ch09.DocumentSerializer
ingest.sinks.ingestHBaseSink.channel = ingestKafkaChannel
```

拦截器

Flume拦截器是一段代码，可以在Flume事件被送到sink之前更新或丢弃它。

在我们的例子中，我们想将XML源转换为Avro对象并执行一些HBase查询以便在发送回HBase之前将其完善。事实上，如果名字或姓氏丢失了，我们需要在HBase中对于已经存在的事件执行查找，以查看是否可以使用该信息完善当前事件。

拦截器可用于这类对象转换和信息完善。这个进程应尽可能快地执行完成，以便可从HBase表中将事件迅速返回到Flume中。如果花费太多时间或在拦截器中执行太多处理过程，将导致Flume处理事件的速度跟不上事件产生的速度，而且可能会给channel 队列产生巨大的压力。

拦截器的配置与source、sink和channel类似：

```
ingest.sources.ingestKafkaSource.interceptors = ingestInterceptor
```

```
ingest.sources.ingestKafkaSource.interceptors.ingestInterceptor.type = \
                              com.architecting.ch09.DocumentInterceptor$Builder
```

可以用Morphline拦截器进行转换，而不需要构建自己的XmlToAvro拦截器。但是，简单起见，为了能够更新HBase记录并且不需要了解Morphlines所有的细节，我们选择实现我们自己的Java拦截器。

转换。拦截器的第一步是将事件转换为Avro对象。像之前章节一样，我们需要定义一个Avro结构：

```
{"namespace": "com.architecting.ch09",
"type": "record",
"name": "Document",
"fields": [
    {"name": "sin", "type": "long"},
    {"name": "firstName", "type": "string"},
    {"name": "lastName", "type": "string"},
    {"name": "comment", "type": "string"}
 ]
}
```

相关的Java类也是以同样的方式生成的：

```
java -jar ~/ahae/lib/avro-tools-1.7.7.jar compile schema\
                                ~/ahae/resources/ch09/document.avsc ~/ahae/src/
```

与之前章节中已完成的代码类似，此代码可用于序列化并反序列化Avro对象。其中，我们将使用XPath方法来解析XML文档并对Avro对象进行填充。例9-1中的代码是从GitHub存储库的完整示例中提取出来的，展示了如何提取这些XML字段。

例 9-1：XML extraction

```
expression = "/ClinicalDocument/PatientRecord/FirstName";
nodes = getNodes(xpath, expression, inputSource);
if (nodes.getLength() >0) firstName = nodes.item(0).getTextContent();

expression = "/ClinicalDocument/PatientRecord/LastName";
inputAsInputStream.reset();
nodes = getNodes(xpath, expression, inputSource);
if (nodes.getLength() >0) lastName = nodes.item(0).getTextContent();

expression = "/ClinicalDocument/PatientRecord/SIN";
inputAsInputStream.reset();
nodes = getNodes(xpath, expression, inputSource);
if (nodes.getLength() >0) SIN =
    Long.parseLong(nodes.item(0).getTextContent());
```

```
expression = "/ClinicalDocument/MedicalRecord/Comments";
inputAsInputStream.reset();
nodes = getNodes(xpath, expression, inputSource);
if (nodes.getLength() >0) comment = nodes.item(0).getTextContent();
```

查找。拦截器的第二步是检查姓氏和名字字段，以验证它们是否包含所需的信息，如果不包含，则执行HBase查找，来尝试找到此信息。在我们通过HBase找到信息之前，HBase可能包含许多没有客户信息的记录，阅读所有这些记录将消耗时间和资源，而且针对这些信息缺失的记录我们只能这么做（阅读所有这些记录）。由于想保持所有的拦截器尽可能快的操作，如果我们寻找的信息记录在HBase中不是第一条记录，还将更新第一条记录，以供后续调用获得更快的查询速度。

图9-4中表示key 为12345的数据，因为文档1到4没有任何姓或者名信息，当再从Flume中接收到没有任何姓名信息的文档6后，拦截器将执行一个HBase查找，逐行扫描并一个一个的读取列，将会发现仅需要来自文档5的信息。然后它将不得不更新文档1，以确保同一行的下一次插入不必查找所有文档1到文档5的姓名信息，只需从右边的第一个条目中找到它。

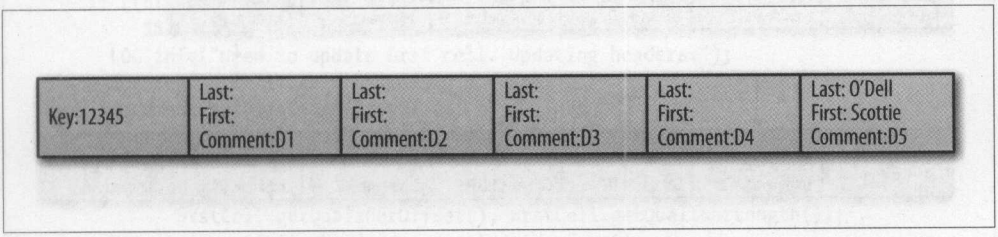


图9-4：以前插入的文档的向后更新

该操作的副作用就是从Flume拦截器中会对HBase执行put操作，这是我们要避免的。因为我们已经有了一个sink用于需要我们更新的HBase表，为什么不简单的从Flume拦截器输出两个事件（一个用于更新的单元格，另一个用于新插入的单元格）？Flume不允许这样做。在Flume拦截器上，你应该输出与接收数量完全相同或少于其的事件。然而，在序列化那一端，可以只接收一个的事件但在同一行中生成两个输入。因此，在拦截器端，如果需要为给定行更新附加列，我们则将相关信息添加到Flume事件header中。该信息将转交给负责实际事件的投放的Flume序列化器。基于此信息，序列化器将为同一行生成一个额外的put操作，这个操作将更新所需的单元格。图9-5展示了这些操作的顺序。

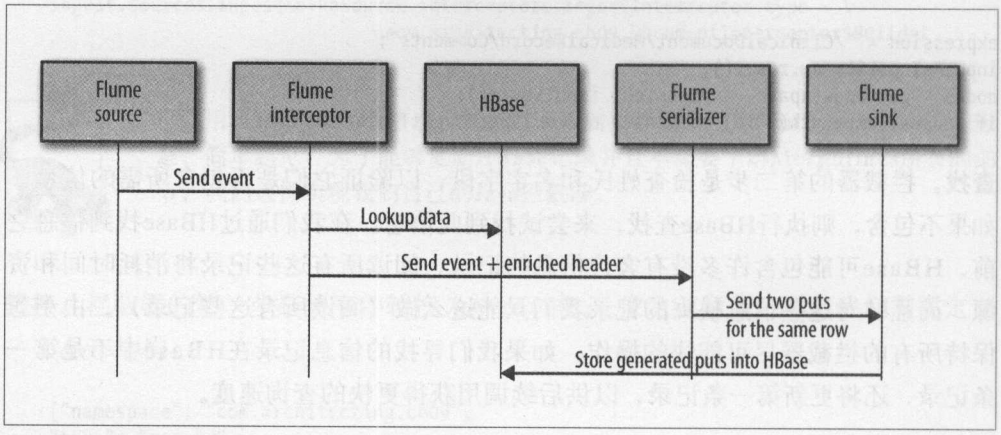


图9-5：操作流程

在所有这些操作之后，数据应该如图9-6所示。

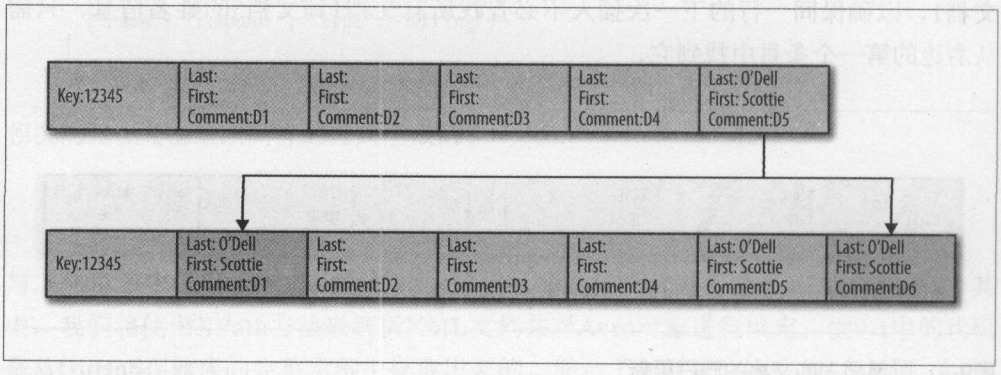


图9-6：拦截器结果

根据你的用例，这可能不是最佳的。事实上，为了避免查找，名字和姓氏字段可能已被提取并插入到各自的单元格中。然而，我们仍然这样做的原因有以下几个方面。首先，这使我们能够说明如何正确使用拦截器和序列化器来和HBase通信，从单个Flume事件中查找并生成多个HBase put。第二，将第一个和最后一个姓名合并同一个单元格中作为另外的字段，使得你可以在读取时通过一次调用获得所有的信息，而不必执行get文档然后再做一次get来获取个人信息。第三，我们在这里查找姓氏和名字，但你也可以从ID查询医生姓名（或关于诊所的任何信息，处方等）再次完善文档本身，不必在读取时执行此查找。因此，为了尽可能提供最多样化和完整的例子，我们决定仍然采用这个表设计。

例9-2展示了如何对Flume执行这些不同的操作拦截器。

例9-2：HBase查找

```
byte[] rowKeyBytes =
    Bytes.add(DigestUtils.md5("" + SIN), Bytes.toBytes((int) SIN));
if (StringUtils.isBlank(firstName) || StringUtils.isBlank(lastName)) {
  LOG.info("Some personal information is missing. Lookup required");
  Table documentsTable = connection.getTable(tableName); ❶
  Get get = new Get(rowKeyBytes);
  Result result = documentsTable.get(get); ❷
  if (!result.isEmpty()) {
    result.advance();
    Cell firstCell = result.current();
    Cell currentCell = firstCell;
    while (true) {
      document = cellToAvro(currentCell, document);
      if (document.getFirstName() != null) firstName =
          document.getFirstName().toString();
      if (document.getLastName() != null) lastName =
          document.getLastName().toString();
      if((!"".equals(firstName) && !"".equals(lastName))
          || (!result.advance())) break; ❸
      currentCell = result.current();
    }
    if ((firstCell != currentCell) && StringUtils.isNotBlank(lastName)
        && StringUtils.isNotBlank(firstName)) { ❹
      LOG.info("Need to update first cell. Updating headers.");
      document = cellToAvro(firstCell, document);
      document.setFirstName(firstName);
      document.setLastName(lastName);
      byte[] qualifier =
          Bytes.copy(firstCell.getQualifierArray(),
            firstCell.getQualifierOffset(), firstCell.getQualifierLength());
      Map<String, String>headers = event.getHeaders();
      headers.put(COLUMN, Base64.encodeBytes(qualifier));
      headers.put(PAYLOAD, Base64.encodeBytes(avroToBytes(document))); ❺
    }
  }
}
```

❶ 为了提高性能，重复使用一个单独的连接来检索表是很重要的。

❷ 获取当前客户信息行，并准备在其所有文档中进行迭代。

❸ 迭代，直到我们找到客户的姓名。

❹ 如果我们找到我们正在寻找的信息，而它不是来自行的第一单元，则更新第一单元以加快后续请求的速度。

❺ 我们决定将整个文档放入标题中。这样做允许我们不必在序列化那一端重新查找表，而只是直接创建相关的Put对象。如果你的文件太大，你可能想避免这样做。

代码的最后一部分，如例9-3所示，是确保修改后的信息被应用回Avro对象，该对象将被序列化并发送回Flume。

例9-3：Avro配置

```
document.setFirstName(firstName);
document.setLastName(lastName);
document.setSin(SIN);
document.setComment(comment);
```

所有这些class都将由Flume Agent运行。因此，对于Flume来说，访问JAR项目很重要。Flume有一个plugin.d目录，每个项目应该创建自己的具有特定结构的子目录。如何构建此目录，Flume文档提供了更多相关的详细信息。

与你构建其他示例方式相同（使用Eclipse或命令行），使用Maven构建这些class的方法将生成的JAR复制到Flume 的plugin目录中。为了简化示例的处理过程，我们将给大家全部可以访问这个目录（在生产环境中，你应该确保只有需要的人才可以访问它）的权限：

```
sudo mkdir -p /var/lib/flume-ng/plugins.d/ahae/lib
sudo chmod a+rwx /var/lib/flume-ng/plugins.d/ahae/lib
cp ~/ahae/target/ahae.jar /var/lib/flume-ng/plugins.d/ahae/lib/
```

你将需要重新启动此Flume以供应用程序考虑该JAR。还必须在每次部署新版本的JAR时必须重新启动Flume agent。

序列化器

Flume序列化器是一个基于给定Flume事件，为给定sink序列化所要求输出的类。这个序列化器将接收我们的由拦截器生成的Avro对象加上所需的header，并将其转换为一个HBase Put操作。根据我们将在header上收到的内容，我们的单次put操作可能会影响一两个列。

HBase Flume序列化程序必须实现*org.apache.flume.sink.HBase.HBaseEventSerializer*接口。该接口需要实现例9-4所示的方法。

例9-4：Flume HBase序列化接口

```
/**
 *初始化事件序列化程序
 * @param写入 HBase的事件
 */
public void initialize(Event event, byte[] columnFamily);

/**
 * Get the actions that should be written out to HBase as a result of this
 * event. This list is written to HBase using the HBase batch API.
 * @return List of {@link org.apache.hadoop.HBase.client.Row} which
 * are written as such to HBase.
 *
 * 0.92 increments do not implement Row, so this is not generic.
 *
 */
public List<Row>getActions();

public List<Increment>getIncrements();
/*
 *清理任何状态。 当sink停止时调用。
 */
public void close();
```

在我们的例子中，我们将实现`initialize`方法和`getActions`。我们不会在`close`方法上做任何具体的事情，因为我们不需要，所以也不需要计数我们的事件，`getIncrements`也将一直为空。例9-5展示了我们的序列化器代码。

例9-5：XML到HBase Flume序列化器

```
@Override
public void initialize(Event event, byte[] cf) {
  if (LOG.isDebugEnabled()) LOG.debug("Performing initialization for a "
      + event.getBody().length + " bytes event");
  else LOG.info("Performing initialization for an event");
  this.payload = event.getBody(); ❶
  this.cf = cf;
  this.headers = event.getHeaders();
  reader = new SpecificDatumReader<Document>(Document.class);
}

@Override
public List<Row>getActions() throws FlumeException {
if(LOG.isInfoEnabled()) LOG.info("Retrieving actions.");
List<Row>actions = new LinkedList<Row>();
try {
  decoder = DecoderFactory.get().binaryDecoder(payload, decoder);
  document = reader.read(document, decoder); ❷
  byte[] rowKeyBytes =
      Bytes.add(DigestUtils.md5("" + document.getSin()),
        Bytes.toBytes(document.getSin().intValue()));
  LOG.info("SIN = " + document.getSin());
  LOG.info("rowKey = " + Bytes.toStringBinary(rowKeyBytes));
```

```
    Put put = new Put(rowKeyBytes);
    put.addColumn(cf, Bytes.toBytes(System.currentTimeMillis()), payload); ❸
    actions.add(put);
    String firstCellColumn;
    if ((firstCellColumn = headers.get(COLUMN)) != null) {
      String payload = headers.get(PAYLOAD);
      put.addColumn(cf, Base64.decode(firstCellColumn), Base64.decode(payload)); ❹
      LOG.info("Updating first cell "
          + Bytes.toStringBinary(Base64.decode(firstCellColumn)));
      }
    } catch (Exception e) {
      LOG.error("Unable to serialize flume event to HBase action!", e);
      throw new FlumeException("Unable to serialize flume event to HBase action!",
          e);
      }
    return actions;
  }
```

❶ 存储payload，列族名称和标题以便以后重复使用它们。

❷ 从接收的有效内容构造document Avro对象。

❸ 创建要发送到表所需的HBase Put对象。

❹ 如果需要，使用更新的信息添加现有Put对象的第二列以覆盖第一列单元格。

在拦截器和序列化器步骤结束时，Flume HBase sink将接收从序列化器生成的put，并将其应用于配置的表。到目前为止，所有的步骤可以单独运行。 你不需运行Lily Indexer或Solr服务。 如果你尝试运行它们，你应该能够看到由你生成的，输入Kafka队列中数据、进入Flume channel Kafka队列的中间数据，以及你在HBase表中的转换数据，以上三种数据将需要由你来创建。

以下命令将帮助你了解整个创建过程。

首先，使用以下命令，查找来为整个处理过程source的Kafka topic信息：

```
kafka-console-consumer --zookeeper localhost/kafka --topic ingest \
                       --from-beginning
```

这将显示那些你插入到topic中的，以供Flume挑选的所有事件。要启动和简化调试，请确保只插入少数几个event。

如果Flume agent正在运行，并且Flume source配置正确，你应该也可以看到哪些事件被筛选出来，并且被插入到Flume Kafka channel topic中。要检查特定的topic，请使用以下命令：

```
kafka-console-consumer --zookeeper localhost/kafka --topic flumechannel\
                       --from-beginning
```

如果一切正常运行，这个topic的内容应该与摄入topic的内容是相同的，因此这两个命令的输出应该是一样。在这一点上，你就可以确认了你的Kafka服务和你的Flume服务正在正常运行。下一步是验证拦截器和序列化器正常运行。 因为Flume sink是HBase sink，我们操作的结果应该在HBase中可见。在HBase shell中对以前创建的表执行以下scan操作，应该会显示与摄入的Kafka topic中事件一样多的单元格：

```
hbase(main):001:0> scan 'documents'
```

如果你已将数百万个事件插入到Kafka输入topic中，在两个topic中将有数百万条目，同时数百万条目的行将会插入到HBase中。 如果是这种情况的话，请务必在与`kafka-console-consumer`命令结合使用时，使用HBase scan的LIMIT关键字以及`more`或`less`。

在*resources/ch09/*文件夹下的*produce.sh*文件将会产生五个事件（其中四个没有姓名，一个具有姓名）。 在进程的最后，你应该看到HBase正在更新的第一个单元，第四个事件被自己的名字和姓氏填充，并且第五个事件被其他事件的姓氏和名字完善。

在不同的运行环境中，有些步骤可能会失败。 如果你没有在摄取topic中看到自己的数据，检查日志和数据生成步骤的代码，然后尝试从命令行推送一些数据。如果你在Flume channel topic中没有看到你的数据，你的Flume agent可能未运行或配置不正确。检查你的Flume agent日志及其配置。 如果你在HBase表没有看到你的数据，可能是你的拦截器或者是你的序列化代码出现了问题。检查你的Flume agent日志和你的Java代码，并确保它已被部署并且能被Flume访问。

HBase

即使HBase可能是本设计最重要的部分，但它的实施却是（工作量）最小的一个。事实上，即使这里的一切都与HBase有关，其实没有任何特定的HBase代码需求。在HBase方面，有两件事我们将不得不研究。

表设计

像往常一样，第一个也是最重要的设计是表和key的设计。如前所述，为了让我们的key有一个合理的分布，我们将使用社会保险号码的MD5哈希值作为key的前缀，接

着后面跟着数字本身。一个MD5哈希值是由一个字节数组表示，这些字节值在0至255之间。有了这些信息，我们才能够正确对表进行预分割，并选择正确的逻辑来计算分割点。因为我们要去生成二进制key，这里将使用uniformsplit算法。我们将创建的region个数取决于群集的大小和我们所预期拥有的数据集的大小。对于我们的示例，我们将创建一个拥有8个region的表，名称叫做document和一个名为C的列族：

```
create 'documents', { NUMREGIONS => 8, SPLITALGO => 'UniformSplit' }, \
                                { NAME => 'c' }
```

预先分割表

正如我们很多次建议的一样，推荐预先将你的表成多个region。这是为了避免RegionServer插入时热点故障，通过减少大的region分割和合并将会有所帮助。然而，一开始设置多少个region？可以从两个很好的数字开始。第一个是基于你的表完全加载后的预期大小。如果数据摄入会发生在很长一段时间（说几个月），考虑几周之后的大小。确保你每个region至少分配1GB。

因此，在一个含20 RegionServer集群，如果你预测你的表在一个月后只有15 GB，开始只需创建15个split。第二个数字是基于你的RegionServer的个数。随着你的数据变得更为分散，读和写会变得更有效。再次，在相同的20 RegionServer集群，对于一个特定的表，你可能希望region个数至少是你RegionServer个数的两倍。如果你的集群主机只有很少的表（两个或三个），你可能希望每个表至少有四个region。然而，要确保永远尊重第一个建议，就是确保你每个region至少分配1GB。

另一个例子，如果你在一个20 RegionServer集群上拥有一个30 GB的表，基于RegionServer的数量，你的表要有40个region。但这将创建1GB以下的region，这是需要我们避免的。因此，我们将使用按照我们预分割的分裂因子计算出来的大小，这也会将这个表预分裂成30个region。

表参数

HBase设计的第二个重要的部分是表参数的设置。的确，正如我们已经看到的，我们可以对HBase表中许多不同的值和参数进行调整。首先，我们要启用XML的压缩。由于XML是文本为主，一般情况下有很好的压缩性能。同时，因为SLA相关项

目的读取延迟很高（文档需要在一秒钟内返回），我们将选择一个和其他算法相比可以提供更好的压缩比的压缩算法。

要启用此表列族的压缩，使用以下命令：

```
alter 'documents', { NAME => 'c', COMPRESSION => 'GZ' }
```

除了压缩之外，同时要禁用布隆过滤器。所有对HBase表的读取使用都将基于Solr中返回来的key。因此，它们总会成功。如果单个region中有许多文件，布隆过滤器可能仍然是有用的。然而，大合并被安排每天晚上定时执行，大多数region将只有一个文件，布隆过滤器将成为一个系统开销。

如果不确定是否使用布隆过滤器，最好是将其保持启用。在几乎不使用的情况下，建议禁用它们。如果你的读取总是成功的（例如，当它们被从外部索引查出来），或如果有每个region只有一个文件（或极少数文件），那么布隆过滤器可以被禁用。然而，对于所有其他使用情况下，布隆过滤器将有助于跳过一些文件的读取，并会改善读取延迟。

使用以下命令，禁用Bloomfilter：

```
alter 'documents', { NAME => 'c', BLOOMFILTER => 'NONE' }
```

如果你保持布隆过滤器开启并禁止压缩，这个例子也将正常运行。

最后，你的HBase表看起来应该像这样：

```
hbase(main):021:0> describe 'documents'
Table documents is ENABLED
documents
COLUMN FAMILIES DESCRIPTION
{NAME => 'c', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'NONE',
REPLICATION_SCOPE => '0', VERSIONS => '1', COMPRESSION => 'GZ',
MIN_VERSIONS => '0', TTL => 'FOREVER', KEEP_DELETED_CELLS => 'FALSE',
BLOCKSIZE => '65536', IN_MEMORY => 'false', BLOCKCACHE => 'true'}
1 row(s) in 0.0290 seconds
```

Java实现

也可以使用Java API创建相同的表。如果你想在版本控制系统中跟踪如何创建自己的表，这可能是非常有用的。例9-6展示了如何创建一个单列族表，其所需的分裂和配置参数与我们先前使用shell方式是一致的。

例9-6：Java创建表的代码

```
TableName documents = TableName.valueOf("documents");
HTableDescriptor desc = new HTableDescriptor(documents);
HColumnDescriptor family = new HColumnDescriptor("c");
family.setCompressionType(Algorithm.GZ);
family.setBloomFilterType(BloomType.NONE);
desc.addFamily(family);
UniformSplit uniformSplit = new UniformSplit();
admin.createTable(desc, uniformSplit.split(8));
```

Lily

配置Lily是很容易的。我们只需给它配置一个morphline脚本，用于描述如何处理HBase的单元格以及对应的push路径：

```
SOLR_LOCATOR : {
  # Name of solr collection
  collection : DocumentCollection

  # ZooKeeper ensemble
  zkHost : "$ZK_HOST"
}


morphlines : [
{
  id : morphline
  importCommands : ["org.kitesdk.**", "com.ngdata.**"]

commands : [
  { logInfo{ format : "Getting something from HBase: {}", args : ["@{}"] } }
  {
    extractHBaseCells {
      mappings : [
      {
        inputColumn : "c:*"
        outputField : "_attachment_body"
        type : "byte[]"
        source : value
      }
      {
        inputColumn : "c:*"
        outputField : "documentId"
        type : "string"
        source : qualifier
      }]
```

```
    }
  }
  { logInfo{ format : "Reading Avro schema: {}", args : ["@{}"] } }
  {
    readAvro {
      writerSchemaFile : /home/cloudera/ahae/resources/ch09/document.avsc
    }
  }
  { logInfo{ format : "Extracting Avro paths: {}", args : ["@{}"] } }
  {
    extractAvroPaths {
      paths : {
        sin : /sin
        firstName : /firstName
        lastName : /lastName
        comment : /comment
      }
    }
  }

  { logInfo{ format : "output record: {}", args : ["@{}"] } }
  {
    sanitizeUnknownSolrFields
    {
      solrLocator : ${SOLR_LOCATOR}
    }
  }
 ]
}
]
```

对Lily而言，就是需要我们提取列限定符，并将其存储到`documentid`字段，然后提取单元格内容为Avro对象，以及提取我们希望复制的特定字段。

我们还需要一个XML文件来配置索引。这个文件将作为命令行一个参数。它定义了我们想使用的mapper类型以及提供了该mapper的一些参数：

```
<?xml version="1.0"?>
<indexer table="documents"
mapper="com.ngdata.HBaseindexer.morphline.MorphlineResultToSolrMapper"
mapping-type="column" row-field="rowkey">

<!-- The relative or absolute path on the local file system to the morphline -->
<!-- configuration file. Use relative path "morphlines.conf" for morphlines -->
<!-- managed by Cloudera Manager -->
<param name="morphlineFile" value="morphlines.conf"/>

</indexer>
```

XML将通过配置Lily来使用`MorphlineResultToSolrMapper`。同时，它也将配置此mapper来在列级别处理HBase的单元格（而不是默认行级别模式）。最后，它还将

通过配置Lily来提供HBase row key到rowkey字段。关于XML文件内容的更多信息可在NGDATA GitHub页面获得。

最后一步是使用以下命令来使得索引器生效：

```
hbase-indexer add-indexer --name myIndexer --indexer-conf indexer-config.xml \
                          --connection-param solr.zk=quickstart/solr \
        --connection-param solr.collection=DocumentCollection \
        --zookeeper localhost:2181
```

如果你想禁用和删除索引（因为你想尝试另一种方式或者存在配置错误），使用以下命令：

```
hbase-indexer delete-indexer --name myIndexer
```

这一步之后，Lily将通过Flume为每个已插入HBase的单元格创建一个Solr document。然而，由于Solr尚未配置，这可能会失败。继续到下一节中去配置Solr并测试Lily。

Solr

Solr实现部分很简单。我们只需要基于我们先前定义的schema创建一个Solr集合。为了创建Solr集合，我们将遵循与第7章中描述的步骤即可。

使用下面的命令创建Solr集合。

```
export PROJECT_HOME=~/ahae/resources/ch09/search
rm -rf $PROJECT_HOME
solrctl instancedir --generate $PROJECT_HOME
mv $PROJECT_HOME/conf/schema.xml $PROJECT_HOME/conf/schema.old
cp $PROJECT_HOME/../schema.xml $PROJECT_HOME/conf/
solrctl instancedir --create DocumentCollection $PROJECT_HOME
solrctl collection --create DocumentCollection -s 1
```

当这些命令都执行完后，应该在Web UI看到Solr集合：*http://quickstart.cloudera:8983/solr/#/DocumentCollection_shard1_replica1*。

测试

现在，所有的部分都安装完成并已成功实现，任何插入到Kfaka队列的数据在HBase和Solr都是可见的。如下的一个Solr查询应该返回所有被该Solr索引过的记录：*http://quickstart.cloudera:8983/solr/DocumentCollection_shard1_replica1/select?q=%*

3A&wt=json&indent=true。所有这些记录也应被插入到HBase中，并且也可以通过shell查看。

插入此XML：

```
<?xml version=\"1.0\" encoding=\"UTF-8\"?>
<ClinicalDocument>
  <PatientRecord>
    <SIN>12345</SIN>
    <FirstName>Scottie</FirstName>
    <LastName>O'Dell</LastName>
  </PatientRecord>
  <MedicalRecord>
    <Comments>Scottie is doing well.</Comments>
  </MedicalRecord>
</ClinicalDocument>
```

将在HBase生产这样的条目：

```
hbase(main):002:0> get 'documents', '827c12345'
COLUMN          CELL
c:1448998062581 timestamp=1448998062683
                value=\xF2\xC0\x01\x0EScottie\x0CO'Dell6Scottie is doing well.
1 row(s) in 0.0290 seconds
```

在Solr中的输出：

```
{
  "responseHeader":{
    "status":0,
    "QTime":0,
    "params":{
      "indent":"true",
      "q":"rowkey: \"827c12345\"",
      "wt":"json"}},
  "response":{"numFound":1,"start":0,"docs":[
    {
      "id":"827c12345-c-1448998450634",
      "lastName":"O'Dell",
      "rowkey":"827c12345",
      "firstName":"Scottie",
      "documentId":"1448998450634",
      "_version_":1519384999917256704}]
}}
```

现在你可以通过任何的索引字段查询Solr，根据使用返回的信息，在给定特定行与特定列的位置的情况下来查询HBase，并将获得一个直接的、低延迟的随机入口以访问记录。

进一步

如果你想扩展本章中提出的例子，下面的列表提供一些选项，你可以基于我们本章中的讨论做些尝试：

更大的输入文件

在这里，我们再次在一个时间摄取一个文件来尝试我们的工作流只有少许。尽量摄取更多的文件，也尽量摄取那些字段比较大的文件。当产生大于1MB的文件时，甚至还可以尝试使用HBase的MOB特征。

提取病历

从文档中提取病人的名称，并将其分别存储在一个特定的列中。只有当信息缺失时修改查找来填充此列。然后，只存储评论字段，而不是将Avro对象存储在HBase。

重复的流

在两个不同的主题和两个Flume agent中生成两种文件。将这两个流合并到一个的HBase表和一个Solr索引中。

行键

一个新文档的行键计算两次（一次在拦截器中，一次在序列化器中）。使用事件header、存储、传输并重新使用该信息。

改变Solr schema

为了让事情变得更简单，在当前的shema中，我们存储、索引和返回行行键。然而，Solr document ID（存储在ID字段）已包含row key以及列族和列限定符。更新Solr模式只对rowkey和document ID 构建索引但不存储它们。

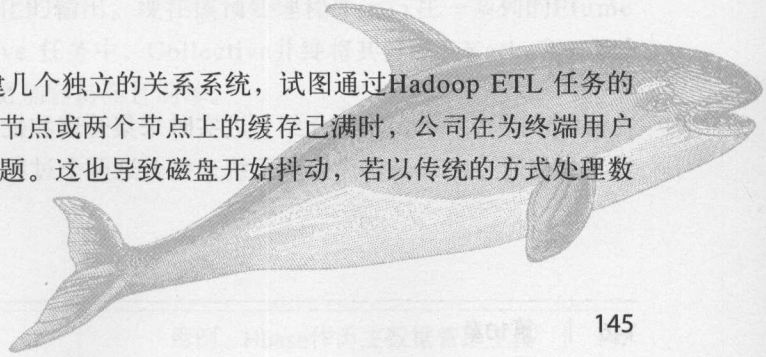
第10章

用例：HBase作为主数据管理工具

接下来，我们看看纽约的数字广告Collective公司，该Collective公司是一家广告技术公司。通过分析品牌与客户之间的关系来提高公司的影响力，这种方法被称为360度全方位客户关系，为了能够有效的执行，它需要各种数据源及大量的数据，所以Collective借助Hadoop和HBase来为他们的客户提供统一的消费视图，并且通过相应渠道及设备(含个人电脑、平板电脑以及智能手机)，实现与品牌信息的无缝传递。

关于360全方位客户关系，Hadoop是一个显而易见的选择。一个可信赖的360全方位客户关系将随着数据源的增加变得越来越好，而且客户定期交互的数据源往往会形成非结构化数据。在360全方位客户关系中，我们会看到一些较为典型的数据源，如通过Adobe Omniture 或 IBM Tealeaf 产生的单击流，以及直接通过Twitter 和Facebook 产生的社交媒体数据，或更多的时候来自于策划的数据服务提供商如Gnip和DataSift。这些数据后续会与一些综合的(含本地)用户属性进行整合，保持持续更新，并在一个类似HDFS的持久文件系统中，通过一些简单的任务完成用户属性的整合，Collective需要一个能够整合大量数据源的系统，使得其能够将零星分散的数据整合到一个关系系统中，形成一个统一的客户视图，而这正是HBase所能够做到的。

首先，Collective曾尝试着构建几个独立的关系系统，试图通过Hadoop ETL 任务的方式去推送数据。但是当一个节点或两个节点上的缓存已满时，公司在为终端用户提供数据服务的时候遇到了问题。这也导致磁盘开始抖动，若以传统的方式处理数



据则需要大量的专业硬件检修，大多数这些关系型系统只能进行垂直的可扩展性，而不是水平的可扩展性，而且通常也需要更多的内存和昂贵的固态硬盘(SSD)；而在另一方面，HBase只需要使用普通的硬件和SATA硬盘，这也是Collective开始转向HBase的原因所在。幸运的是，Collective已经有一个Hadoop集群在使用，只需进行小量的开发及成本代价即可无缝整合HBase到现有的基础架构中。

Collective目前部署了60个RegionServers，仅HBase存储的数据量就达到了20 TB，而且HBase的部署过程也极其简单，在这个用例中，有一个单独的表来处理用户属性数据，该表划分为visitor、export以及 edge三个列族。visitor列族包含用户的元数据，其中涉及出生日期，行为信息和相关第三方查找的ID信息；export列族包含特征属性信息(如男、25岁、喜欢汽车等特征)以及相关的下游需要处理的聚合信息；edge列族包含用户的活动信息以及从批导入的数据中可能出现的任何额外数据。

```
COLUMN CELL

edge:batchimport ts=1391769526303, value=\x00\x00\x01D\x0B\xA3\xE4.
export:association:cp ts=1390163166328, value=6394889946637904578
export:segment:13051 ts=1390285574680, value=\x00\x00\x00\x00\x00\x00\x00\x00
export:segment:13052 ts=1390285574680, value=\x00\x00\x00\x00\x00\x00\x00\x00
export:segment:13059 ts=1390285574680, value=\x00\x00\x00\x00\x00\x00\x00\x00
...
visitor:ad_serving_count ts=1390371783593, value=\x00\x00\x00\x00\x00\x00\x1A
visitor:behavior:cm.9256:201401 ts=1390163166328, value=\x00\x00\x00\x0219
visitor:behavior:cm.9416:201401 ts=1390159723536, value=\x00\x00\x00\x0119
visitor:behavior:iblocal.9559:2 ts=1390295246778, value=\x00\x00\x00\x020140120
visitor:behavior:iblocal.9560:2 ts=1390296907500, value=\x00\x00\x00\x020140120
visitor:birthdate ts=1390159723536, value=\x00\x00\x01C\xAB\xD7\xC4(
visitor:retarget_count ts=1390296907500, value=\x00\x00\x00\x00\x00\x00\x00\x07
```

如前所述，Collective是一家数字广告公司，它通过消费者的行为信息来构建客户画像。在进程中通过上游系统产生的自定义cookie ID 来跟踪用户的行为信息。其中row key被设计为反向的cookie ID。关于为什么会使用一个反向的UUID，主要有两个因素：website和时间序列数据。在这种情况下，cookie ID的前置部分包含了timestamp时间戳。这将导致row key 简单递增，但可通过对row key的简单反置，随机生成的部分信息会分布在row key的前部位置。

摄取

在第8章和第9章中，我们介绍了近实时采集管道和批量加载过程。接下来，我们来看看如何通过HBase作为记录系统来整合这两个过程，有时HBase也被称为“主数

据管理MDM”或“黄金副本”数据记录系统。当发生数据损坏或者错误时，HBase中的记录将用于重构Hive或其他外部数据源。其中首先需要检查的是批处理。为了该记录系统，Collective在基于第三方多数据源的基础上构建一个专门的数据拉取工具，其中包括S3，SFTP网站以及部分专用的APIS (见图10-1)等多种数据源。该工具每小时从这些数据源中拉取数据，并以Parquet文件格式加载这些数据到新的HDFS目录中，并在新加载的数据基础之上创建新的hive 分区，最后链接到现有的归档表中。

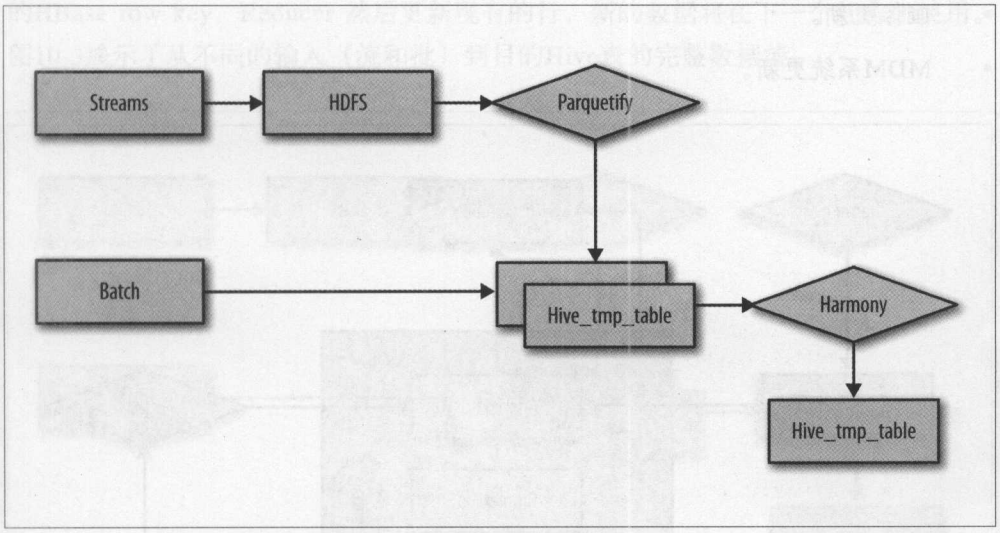


图 10-1：摄取数据流

另外一方面是准实时处理，目前是通过Flume来处理。Flume从不同消息服务中引入数据，然后将数据推送到一个称为Parquetify的系统中。该系统每小时运行一次，首先将数据转化为统一的文件格式，并对该部分数据构建一个临时表，然后将该部分数据插入到一个主的归档hive表中。在这种情况下，文件格式为Parquet格式，一旦数据被加载到系统中，一个被称为Harmony的预处理程序每个小时都将按照Celos自定义工作流运行，Harmony 首先从前面列举的各种数据源中收集并预处理所需的数据，并为正式转化阶段做标准化的输出。现在该预处理程序运行在一系列的Flume数据流、MapReduce任务和Hive 任务中，Collective并终将其推送到Kafka和Spark上，从而可使得整个处理过程更加容易而且简单。

处理

一旦Harmony加入到内部数据整合中，在最后的处理过程中，它将会被推送到一个用于协调一系列MapReduce 任务的另外一个内部系统中，它就是我们所熟知的Pythia (见图10-2)，为了在失败的情况下保持数据的一致性，这些MapReduce 任务实现了write-ahead log (WAL)过程。在这个过程中有三个总的步骤：

- 聚合。
- 画像更新。
- MDM系统更新。

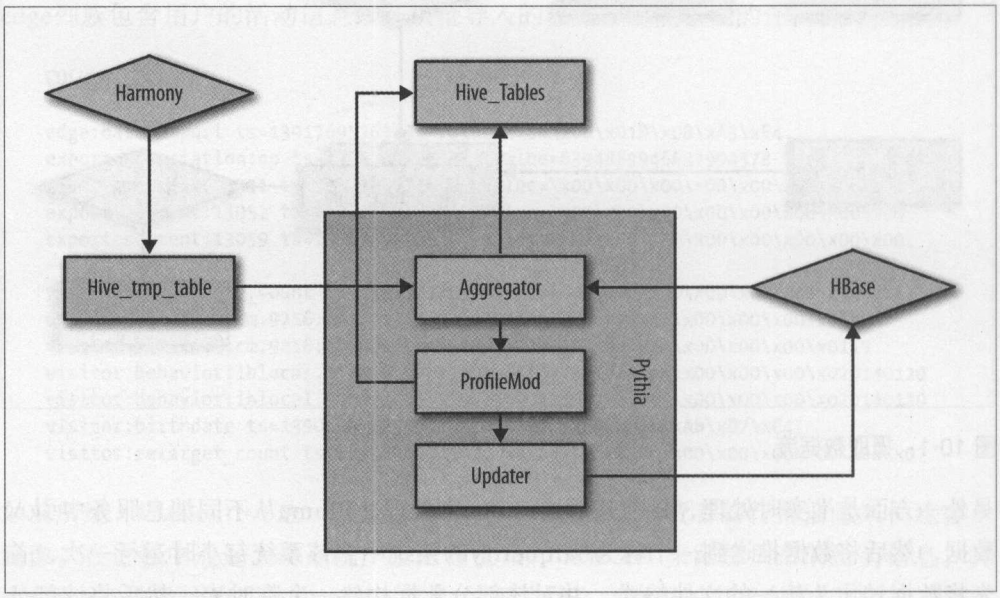


图10-2：处理数据流

在这个过程中，聚合和画像更新都支持分区的Hive/Impala 表，为了聚合任务，首先需要做的事是读取Harmony 任务的输出信息到Avro 文件，只要新的输出数据(Harmony)和前一小时的数据（HDFS）在mapper中都被读取并被分组，它们将会被传输到reducer中进行处理，在Reduce阶段，为了下一个小时的数据分区，将会产生大量的HBase 调用请求来产生一个整体的画像，其中reducer将为已存在的每一条画像记录去拉取整个HBase 记录进行画像重构。集合任务然后将会输出一组差异(通常称为差别）以便于画像更新阶段使用，假如上游系统作业任务失败，这些差别被用作为分类的WAL来纠正这些变更。

接下来，画像更新作业任务开始了，由于在聚合任务的reducer中，我们已经从HBase中抽取出我们需要的数据了，所以画像更新任务将只是一个Map 任务，这个mapper将从先前的输出和校正的差别中读取出所有的数据，只要这些差别都被整合，画像更新任务将使用这些真实的差异回写到HBase中，该任务最后会在画像更新 Hive表中创建一个新的时分区。

最后，记录的MDM系统（HBase）需要得到更新。最后一步是另一个MapReduce的任务，这项任务（更新）读取画像模型数据输出，然后基于之前的数据，构建正确的HBase row key。Reducer 然后更新现有的行，新的数据将在下一个任务中使用。图10-3展示了从不同的输入（流和批）到目的Hive表的完整数据流。

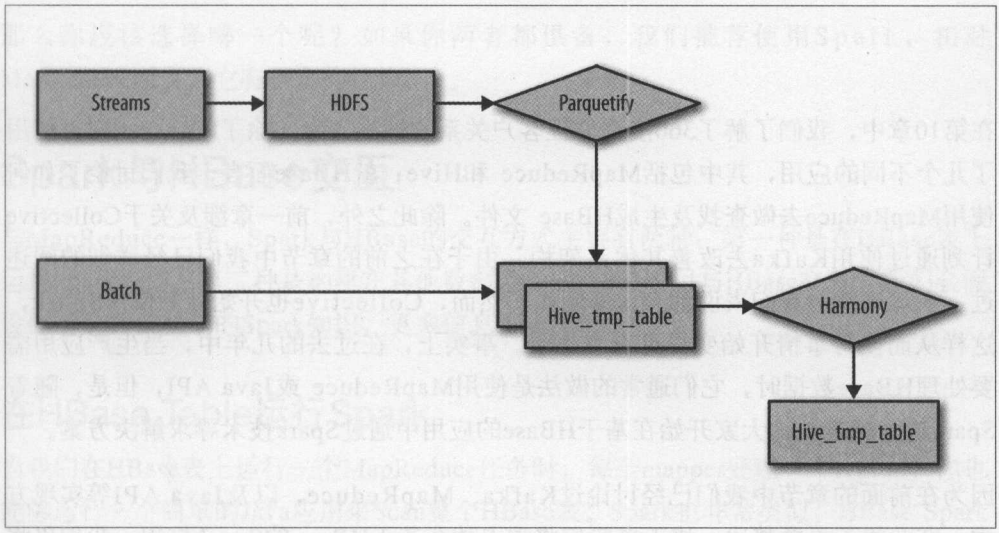


图10-3：完整数据流

第11章

主数据管理工具HBase的实现

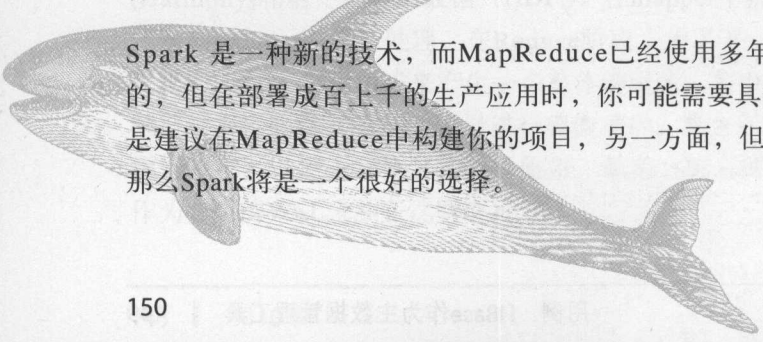
在第10章中，我们了解了360度全方位客户关系的解决方案，除了HBase，它也使用了几个不同的应用，其中包括MapReduce 和Hive；就HBase而言，我们讨论了如何使用MapReduce去做查找及生成HBase 文件。除此之外，前一章涉及关于Collective计划通过使用Kafka去改善其体系架构，由于在之前的章节中我们已经详细的阐述过，所以这些对我们来说都应该不陌生；然而，Collective也开始计划使用Spark，这样从而使得事情开始变得更有意思了。事实上，在过去的几年中，当生产应用需要处理HBase数据时，它们通常的做法是使用MapReduce 或Java API，但是，随着Spark越来越普及，大家开始在基于HBase的应用中通过Spark技术寻求解决方案。

因为在前面的章节中我们已经讨论过Kafka、MapReduce，以及Java API等实现方式，在此将不再做赘述，接下来我们将重点放在基于HBase的Spark应用，我们仍将以构建客户360度全方位画像为实现案例，其他实现案例均可参考。

MapReduce与Spark

在继续之前，我们首先应该明确Spark与MapReduce的利弊，虽然不会为这个话题进行一个冗长的讨论，但我们将简要突出一些要点让你考虑，从而缩小你的选择。

Spark 是一种新的技术，而MapReduce已经使用多年，虽然Spark已被证明是稳定的，但在部署成百上千的生产应用时，你可能需要具备一定娴熟的技能，否则，还是建议在MapReduce中构建你的项目，另一方面，但如果你喜欢引入最新的技术，那么Spark将是一个很好的选择。



针对于已经部署了很多MapReduce 项目，并且具备良好的MapReduce经验积累的公司而言，使用原来擅长的技术实现可能会使项目更容易、更快，而且更便宜，相反，如果开始一个全新的项目或准备计划启动很多项目的话，可能就会考虑采用Spark了。

在一些需要快速高效的数据处理用例中，MapReduce可能是一个大的开销，它在系统的稳定性及其他方面有一定的优势，但性能方面相对较弱。如果你的用例对SLA要求较高，你可以考虑Spark。否则，MapReduce仍然是个不错的选择。

最后一个考虑点是开发语言。Spark与Java和Scala代码兼容，而MapReduce主要针对Java开发者。

那么你应该选择哪一个呢？如果你两者都俱备，我们推荐使用Spark，相对MapReduce而言，它有一定的优势。

Spark与HBase交互

与MapReduce一样，Spark与HBase的交互方式主要有两种；第一种是在HBase表上运行Spark应用；第二种是构建在其他数据之上的Spark应用与HBase交互。我们会假设你至少具备基本的Spark知识，更多信息可参阅Spark官网。

在HBase Table运行Spark

当我们在HBase表上运行一个MapReduce任务时，每个mapper处理一个region，你也能够运行一个简单的Java应用来Scan整个HBase表。Spark也非常类似，HBase Spark API能够返回一个代表整个HBase 表的RDD，但需要你划分该RDD给多个executors去处理，每个executor处理一个region。

从Spark调用HBase

可以从Spark 程序中去访问HBase的 Java API ，因此，你可使用类似Flume拦截器的方式去实现查找，你也能够在Spark程序中，采用Puts和Increments的方式，加工并存储数据到HBase中，最后，与使用MapReduce生成HFile文件，实现BulkLoads功能一样，你也可以使用Spark生成同类文件，然后BulkLoad 进入 HBase中。

Spark结合HBase实现

为了说明Spark与HBase结合方式，首先，我们将实现一个spark job 来处理文本文件，其中为了增加HBase的数据量，设定文本文件的每行都作为一条记录，接下来，我们将另外处理一类源数据，并将其以HFile格式BulkLoad 到HBase 表中，最后在基于相同的HBase表基础上，通过spark job的方式实现数据的汇总。

你可能会发现不同版本的Spark HBase API。但我们使用的是来自于Apache HBase2.0 项目。如果你使用的是另一个版本的Spark HBase API，你可能会面临一些编译问题，但背后的逻辑仍然是相同的。可能需要调整示例来编译代码并运行它们。

我们要执行的所有操作都将在同一个表上完成，使用以下命令创建此表：

```
create 'user', {NAME => 'edge', BLOOMFILTER => 'NONE',\
                COMPRESSION => 'SNAPPY', DATA_BLOCK_ENCODING => 'FAST_DIFF'},\
               {NAME => 'segment', BLOOMFILTER => 'NONE',\
                COMPRESSION => 'SNAPPY', DATA_BLOCK_ENCODING => 'FAST_DIFF'},\
               {NAME => 'visitor', BLOOMFILTER => 'NONE',\
                COMPRESSION => 'SNAPPY', DATA_BLOCK_ENCODING => 'FAST_DIFF'}
```

让我们仔细看看表创建命令和我们选择的不同选项的原因。首先，我们禁用了Bloom过滤器，事实上，我们每次查询HBase的行记录时，都将使用已经确认存在的rowid作为查询条件。此外，大多数时候，整个数据集将被处理，而不仅仅是一个单一的行。因此，必须查看Bloom过滤器，以确认该行是否存在。而且通常情况下，开启Bloom过滤器将会是一个良好的习惯。如果你不确定是否需要开启的话，就尽量开启吧。

其次，我们启用Snappy压缩机制。虽然Snappy相比其他一些算法的压缩比可能是低效率的，但是其允许快速压缩的机制对CPU的影响是非常小的。除非你像存储类似于压缩图像或音频等数据，否则开启Snappy始终是一个好的建议。

最后，我们启用了数据块编码。数据块编码是非常有用的，它能够减少为同一行的每一列存储密钥的开销。考虑到一个用户可以有多个列，以及行键可以相当长，激活FAST_DIFF块编码将为我们节省更多的存储空间。

Spark与HBase: Puts

在处理输入文件时，可通过如下两种方式来丰富HBase数据；第一种方式为实时处理，我们可以在解析输入文件的时候，也可同时在HBase中更新数据，这样一旦Spark读取到数据，就可以在HBase中看到结果。你也可以通过类似的代码或Spark streaming 来实现一个流处理管道。另外，若你没有在HBase中获取实时数据的需求，你可以将输入文件转化成HBase HFile进行批量的加载。相比而言，流处理方式能够使你更实时的更新HBase数据，而批处理方式能够提供更好的吞吐量，因为它不需要与HBase进行交互。

我们首先需要的是一个足够大的文本文件，从而希望能够充分发挥Spark的性能。在后续的Kafka章节中，我们将创建一个小工具来生成这个文件。这个工具可按如下方式进行调用：

```
java -classpath ~/ahae/target/ahae.jar:`HBase classpath`\
com.architecting.ch11.DataGenerator 100000 > ~/ahae/resources/ch11/data.txt
```

文件生成后，必须将其上传至HDFS中进行处理：

```
hdfs dfs -put ~/ahae/resources/ch11/data.txt
```

Spark处理文本文件不需要任何特殊的HBase API，例 11-1 代码简单描述了从HDFS读取一个文件以及处理所有行记录的过程，为了简化实例，我们仅仅实现了统计文件的行数，在后续的阐述中，我们将逐步丰富我们的实例程序。

例 11-1：使用Spark 统计行数的简单实例

```
SparkConf sparkConf = new SparkConf().setAppName("JavaHBaseBulkGetExample ")
                                                    .setMaster("local[2]");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);
JavaRDD<String> textFile = jsc.textFile("hdfs://localhost/user/cloudera/data.txt");
JavaPairRDD<String, Integer> pairs = textFile.
                        mapToPair(new PairFunction<String, String, Integer>() {
  public Tuple2<String, Integer> call(String s) {
    return new Tuple2<String, Integer>(s.substring(0, s.indexOf("|")), 1); }
  });
JavaPairRDD<String, Integer> counts = pairs.
                      reduceByKey(new Function2<Integer, Integer, Integer>() {
  public Integer call(Integer a, Integer b) { return a + b; }
  });
System.out.println ("We have generaged " + counts.count() + " users");
```

这个例子中，我们只是简单的从每一行抽取出用户ID，然后进行汇总实现统计，因为这是一个非常简单的例子，我们只需从Eclipse中使用本地环境简单的运行该程序，尽管DEBUG 日志和INFO 都很长，但你只需关注类似这样的信息：

```
We have generaged 999952 users
```

既然我们的应用程序有了一个大体的骨架，其他一切都将是标准的HBase java API与Spark的调用，正如你可能已经想到的，输入文件中的每个条目代表某人的信息，我们要做的是对这个信息更新到HBase中。如下主要有两种不同的方法实现，第一种方式如例11-2所示，考虑到易于阅读和理解，针对输入文件的每行记录做了一个转化，并将需要发送到HBase表的信息进行缓冲处理。

例11-2：使用Spark执行HBase BulkPut

```
SparkConf sparkConf = new SparkConf().setAppName("IngestLines ")
                                     .setMaster("local[2]");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);
Configuration conf = HBaseConfiguration.create();
JavaHBaseContext HBaseContext = new JavaHBaseContext(jsc, conf);
JavaRDD<String> textFile =
                    jsc.textFile("hdfs://localhost/user/cloudera/data.txt");

HBaseContext.bulkPut(textFile, TABLE_NAME, new Function<String, Put>() {
  @Override
  public Put call(String v1) throws Exception {
    String[] tokens = v1.split("\\|");
    Put put = new Put(Bytes.toBytes(tokens[0]));
    put.addColumn(Bytes.toBytes("segment"),
                  Bytes.toBytes(tokens[1]),
                  Bytes.toBytes(tokens[2]));
    return put;
  }
});
jsc.close();
```

上述代码针对每行以及每列做了相应的Put转化，并简单的将其返回到HBase Spark BuklPut 框架中，如果我们考虑到重复的行，将会发现每行最多只有七列，所以在每行上做一个转换是可以接受的。然而，如果你有几十个列，你可以先通过重组的方式来提高性能，然后创建相关的Puts并直接插入至HBase中。这种方法如例11-3所示，如果你刚接触Spark，示例代码可能更加难以阅读和理解。 然而，在本地虚拟机环境下运行，就数据量在一百万行、7个字段列的情况下， 例11-3代码的运行速度比例11-2性能会提高约10%。如果增加列数，差异会更显著。

例11-3：HBase Spark 处理文本文件

```
SparkConf sparkConf = new SparkConf().setAppName("IngestLines ")
                                     .setMaster("local[2]");
JavaSparkContext jsc = new JavaSparkContext(sparkConf);
Configuration conf = HBaseConfiguration.create();
JavaHBaseContext hbaseContext = new JavaHBaseContext(jsc, conf);
JavaRDD<String> textFile =
                jsc.textFile("hdfs://localhost/user/cloudera/data.txt");
```

```
PairFunction<String, String, String> linesplit = ❶
new PairFunction<String, String, String>() {
  public Tuple2<String, String> call(String s) {
    int index = s.indexOf("|");
    return new Tuple2<String, String>(s.substring(0, index),
                                      s.substring(index + 1));
  }
};

JavaPairRDD<String, String> pairs = textFile.mapToPair(linesplit);
Function<String, List<String>> createCombiner =
new Function<String, List<String>>() {
  public List<String> call(String s) {
    List<String> list = new ArrayList<String>();
    list.add(s);
    return list;
  }
};

Function2<List<String>, String, List<String>> mergeValue =
new Function2<List<String>, String, List<String>>() {
  @Override
  public List<String> call(List<String> v1, String v2) throws Exception {
    v1.add(v2);
    return v1;
  }
};

Function2<List<String>, List<String>, List<String>> mergeCombiners =
new Function2<List<String>, List<String>, List<String>>() {
  @Override
  public List<String> call(List<String> v1, List<String> v2) throws Exception {
    v2.addAll(v1);
    return v2;
  }
};

JavaPairRDD<String, List<String>> keyValues =❷
         pairs.combineByKey(createCombiner, mergeValue, mergeCombiners);

JavaRDD<Put> keyValuesPuts = keyValues.map(❸
new Function<Tuple2<String, List<String>>, Put>() {
  @Override
  public Put call(Tuple2<String, List<String>> v1) throws Exception {
    Put put = new Put(Bytes.toBytes(v1._1));
    ListIterator<String> iterator = v1._2.listIterator();
    while (iterator.hasNext()) {
      String colAndVal = iterator.next();
      int indexDelimiter = colAndVal.indexOf("|");
      String columnQualifier = colAndVal.substring(0, indexDelimiter);
      String value = colAndVal.substring(indexDelimiter + 1);
      put.addColumn(COLUMN_FAMILY, Bytes.toBytes(columnQualifier),
                                   Bytes.toBytes(value));
    }
    return put;
```

```
  }
});

hbaseContext.foreachPartition(keyValuesPuts,❹
  new VoidFunction<Tuple2<Iterator<Put>, Connection>>() {
    @Override
    public void call(Tuple2<Iterator<Put>, Connection> t) throws Exception {
      Table table = t._2().getTable(TABLE_NAME);
      BufferedMutator mutator = t._2().getBufferedMutator(TABLE_NAME);
      while (t._1().hasNext()) {
        Put put = t._1().next();
        mutator.mutate(put);
      }

      mutator.flush();
      mutator.close();
      table.close();
    }
  });
jsc.close();
```

❶ 从行记录中抽取key，然后进行重组生成键值对。

❷ 基于key值划分实体，从而使得每个key都会有一个与之相关联的column qualifier/value 列表。

❸ 转换一个key以及与之相关的column qualifier/value 为一个单一的HBase put对象。

❹ 将所有的put对象都加载入HBase中。

基本上，那些从字符串中抽取出来的key值，正是我们重组所有行的基础，然后将它们转换为对应行的单一put对象发送至HBase中。

你可以直接从Eclipse运行这个示例。

由于配置的方式，此示例只运行两个本地线程，如果你想在YARN上运行它，从代码中删除.set Master("local[2]")参数，然后添加--master yarn-cluster和--deploy-mode客户端参数后再运行一次该示例。

因为这个例子的输出相当冗长，我们不会在这里重现，然而，当它运行结束时，你可以查询你的HBase表以确保数据被处理了：

```
hbase(main):005:0> scan 'user', LIMIT => 2
ROW                COLUMN+CELL
```

```
0000003542a7-... column=segment:postalcode, ts=1457057812022, value=34270
0000013542a7-... column=segment:birthdate, ts=1457057756713, value=20/05/1946
2 row(s) in 0.0330 seconds
```

与我们在第9章中加工数据做的事情类似，在进行转换及数据加工之前，你可以使用默认的HBase API去执行查询操作。

Spark与HBase: Bulk Load

在上一节中，我们讨论了一个实时更新HBase表的方法，如果这样的方式并不是你所需要的，你也可以使用HBase批量加载选项以实现更好的吞吐量。我们以相同的使用场景为例，但这一次，不是直接与HBase互动，我们将会先生成HBase HFiles文件，然后进行上传。

数据将保持不变，主要的区别是在Spark方面，HBase在基于键值边界范围的基础上，分裂表到多个regions中。每一个我们创建的HFile文件都将归属于regions中的一个，并且其包含键值也必须在region边界范围内。

在进行写入操作时，通过Java API的方式进行Spark批量加载时，需要通过JIRA HBASE-14217进行追踪，只有这个JIRA执行完毕，才能够确认数据加载完成。由于只有Scala才可以用于执行此操作，因此，例11-4将使用Scala实现。

例11-4：使用Spark Scala方式实现HBase BulkLoad示例

```
import org.apache.hadoop.fs.Path
import org.apache.hadoop.hbase.mapreduce.LoadIncrementalHFiles
import org.apache.spark.SparkContext
import org.apache.hadoop.HBase.HBaseConfiguration
import org.apache.hadoop.HBase.TableName
import org.apache.hadoop.HBase.client._
import org.apache.hadoop.HBase.spark._
import org.apache.hadoop.HBase.spark.HBaseContext
import org.apache.hadoop.HBase.util.Bytes
import org.apache.spark.SparkConf
import org.apache.hadoop.HBase.spark.HBaseRDDFunctions._

object SparkBulkLoad {
  def main(args: Array[String]) {

    val columnFamily1 = "segment"
    val stagingFolder = "/tmp/user"
    val tableNameString = "user"
    val tableName = TableName.valueOf(tableNameString)

    val sc = new SparkContext(new SparkConf().setAppName("Spark BulkLoad")
                                             .setMaster("local[2]"))
    val config = HBaseConfiguration.create
    val HBaseContext = new HBaseContext(sc, config)
```

```
    val textFile = sc.textFile("hdfs://localhost/user/cloudera/data.txt")
    val toByteArrays = textFile.map(line => {❶
      val tokens = line.split("\\|")
      (Bytes.toBytes(tokens(0)), (Bytes.toBytes(columnFamily1),
                                  Bytes.toBytes(tokens(1)),
                                  Bytes.toBytes(tokens(2))))
    })

    toByteArrays.HBaseBulkLoad(HBaseContext, tableName, ❷
      t => {❸
        val rowKey = t._1
        val family:Array[Byte] = t._2._1
        val qualifier = t._2._2
        val value = t._2._3

        val keyFamilyQualifier= new KeyFamilyQualifier(rowKey, family, qualifier)

        Seq((keyFamilyQualifier, value)).iterator
      },
      stagingFolder)

    val load = new LoadIncrementalHFiles(config)
    load.run(Array(stagingFolder, tableNameString)) ❹
  }
}
```

❶ 基于输入文件的应用转换会将每个行转换为代表密钥、列族、列限定符和值的字节数组。

❷ 基于转换输入，我们将通过Spark HBase API去生成相关的HFile文件。

❸ 描述如何将字节数组转换成批量加载所需的格式。

❹ HFile文件生成后，利用LoadIncrementalHFiles API将生成的文件加载到目标表中。

运行此应用程序需要几个步骤：首先，因为你很可能没有Spark用户运行该应用程序，将需要在applicationhistory目录下授予写权限。你可通过下面的命令进行授权（你可能需要为HDFS设置密码）。

```
sudo -u hdfs hadoop fs -chmod 1777 /user/spark/applicationHistory
```

需要注意的另一个重要的考虑因素，在不安全的集群中，运行HBase LoadIncrementalHFiles需使用HBase用户，事实上，它会把文件推送到*/HBase*目录树下，然后HBase进程从该目录下进行读写；若使用另一个用户推送文件，HBase将不能删除该文件，从而导致进程失败。

你可以使用如下的命令运行一个简单的示例：

```
su -l hbase -- spark-submit --class com.architecting.ch11.SparkBulkLoad \
            -- master local ~/ahae/target/ahae.jar
```

然后增加如下内容到你的HBase表中：

```
hbase(main):013:0> scan 'user', LIMIT =>
ROW                 COLUMN+CELL
  0000123542a7-... column=segment:postalcode, ts=1457567998199, value=34270
  0000153542a7-... column=segment:lastname, ts=1457567998199, value=Smith
  0000173542a7-... column=segment:birthdate, ts=1457567998199, value=06/03/1942
  0000173542a7-... column=segment:status, ts=1457567998199, value=married
```

Spark stream通常被看作是一个微型的批处理引擎，它会一次处理几个条目，然后再处理下一批等机制。在bulk load加载时会产生HFile文件，文件越大，我们的吞吐量就越好。因为每个小的批处理程序都会生成HFile文件，所以使用Spark stream做bulk load也就没有任何意义，我们也就不推荐。

Spark与HBase

最后一个例子大家可以看看如何使用Spark和HBase去处理HBase表数据，前面我们使用了MapReduce进行了HBase表处理，接下来将基于Spark技术做相应的表数据处理。Spark技术理念与MapReduce相似的，它主要并行处理不同的regions。

本章我们需要讨论的示例很简单，它们实现的目标是相同的——统计HBase表的行数。虽然这是一个非常简单的示例，但后面你会看到它如何扩展到更复杂的用例。

我们将把这个例子分解成几个模块，最重要的部分如例11-5所示。

例11-5：基于HBase Spark的HBase初始化

```
// SparkConf sc = new SparkConf().setAppName("ProcessTable").setMaster("local[2]");
SparkConf sc = new SparkConf().setAppName("ProcessTable");
JavaSparkContext jsc = new JavaSparkContext(sc);
Configuration conf = HBaseConfiguration.create();

JavaHBaseContext HBaseContext = new JavaHBaseContext(jsc, conf);

Scan scan = new Scan();
scan.setCaching(100);
KeyOnlyFilter kof = new KeyOnlyFilter();
scan.setFilter(kof);

JavaRDD<Tuple2<ImmutableBytesWritable, Result>> data =
                    hbaseContext.HBaseRDD(TABLE_NAME, scan);
```

这是最有意思的地方，开始是标准的Spark和HBase初始化调用。如果你使用过HBase，正如我们所看到的，当我们在HBase表做MapReduce操作时，我们初始化一个扫描，将我们不需要的行过滤掉，只获取我们需要的行。在这里，因为我们要统计表的行数，所以只需得到所有的行，并且也不需要得到值，只是保留键，这样的话你应该就比较熟悉了。最后一行非常有趣，通过输入相应的表名以及scan对象至HBaseContext实例，你会得到HBase相应整个表的RDD，处理这个RDD也就代表着处理整个HBase表。例11-6为HBase表一个非常基本的统计记录数的操作。

例11-6：Spark HBase RDD统计

```
System.out.println("data.count() = " + data.count());
```

这简单的一行代码将触发Spark RDD计数，这个Spark RDD代表了整个HBase表，它只是统计表中的所有行，并不能真正让你在行上做任何特定的操作。

例11-7和例11-8也是基于HBase表的统计操作，但正如你看到的，尽管他们只是统计记录数，但通过修改这些例子来执行任何的行操作也将是非常容易的。

例11-7：基于HBase Spark 的HBase分区reduce计数

```
FlatMapFunction<Iterator<Tuple2<ImmutableBytesWritable, Result>>, Integer> setup =
new FlatMapFunction<Iterator<Tuple2<ImmutableBytesWritable, Result>>, Integer>() {
  @Override
  public Iterable<Integer>
                    call(Iterator<Tuple2<ImmutableBytesWritable, Result>> input) {
    int a = 0;
    while (input.hasNext()) {
      a++;❶
      input.next();
    }
    ArrayList<Integer> ret = new ArrayList<Integer>();
    ret.add(a);
    return ret;
  }
};
Function2<Integer, Integer, Integer> combine =
new Function2<Integer, Integer, Integer>() {
  @Override
  public Integer call(Integer a, Integer b) {
    return a + b;❷
  }
};

System.err.println("data.mapPartitions(setup).reduce(combine) = "+
    data.mapPartitions(setup).reduce(combine));
long time3 = System.currentTimeMillis();
System.err.println("Took "+(time3 -time2)+" milliseconds");
```

❶ 针对每个Spark partition，我们只需对每行设置一个计数器，就可非常容易的从HBase cell中计算出相应的累积值或者做任何类型你想要的汇总操作。

❷ 现在所有的分区都是聚合的，要汇总起来，我们只需要汇总所有的值。

例11-7看起来非常接近我们在前几章介绍的MapReduce操作，因为它不是真正的通过partition做map，例11-8有点不同，但输出结果是一样的。

例11-8：基于HBase Spark 的HBase 汇总统计count

```
Function2<Integer, Tuple2<ImmutableBytesWritable, Result>, Integer> aggregator =
new Function2<Integer, Tuple2<ImmutableBytesWritable, Result>, Integer>() {
  @Override
  public Integer call(Integer v1, Tuple2<ImmutableBytesWritable, Result> v2)
      throws Exception {
    return v1 + 1;❶
  }
  };
  Function2<Integer, Integer, Integer> combiner =
  new Function2<Integer, Integer, Integer>() {
    @Override
    public Integer call(Integer v1, Integer v2) throws Exception {
      return v1 + v2;❷
    }
  };

  System.err.println("data.aggregate(0, aggregator, combiner) = " +
                                          data.aggregate(0, aggregator, combiner));
  long time4 = System.currentTimeMillis();
  System.err.println("Took " + (time4 - time3) + " milliseconds");
  // end::PROCESS1[]

  jsc.close();
}

public static void main(String[] args) {
  processVersion1();
}

}
```

❶ 由于我们仅想计算行数，我们只为每一行输出1。然而，考虑更复杂的一些事情，想象将HBase cell包含的Avro对象表示订单，如果你想汇总所有订单的金额，这正是你将做的事情，只需从参数提取Avro对象并汇总其金额。

❷ 再次，为了将所有的值整合到一起，我们必须汇总它们。

例11-7和例11-8可以很容易的修改实现方式来处理你的数据：

- 考虑做数据关联应用，可在map端执行HBase查找去增强输出文件内容，并生成Parquet数据来满足你的分析请求。
- 考虑做数据聚集应用，可提升日、周、月的HBase 表数据价值。
- 考虑通过提供Kafka 键的方式允许后续系统接收和处理HBase的数据。

一切你之前通过MapReduce所做的操作，现在都可以通过Spark实现。

进一步

如果想扩展本章中的例子，下面的列表提供了一些选项，可以根据我们的讨论进行尝试：

分区

为了更加清楚了解RDD partition的优势，创建一个更大的含多个regions 的HBase表，并尝试创建两个版本的Spark代码：一个使用一个执行器运行所有数据，另一个对数据进行分区，运行在多个执行器，而不是一个单一的执行者处理整个表，你会看到每个HBase Region都运行一个执行器，这个执行时间差异会非常明显。

流

试着把put操作的例子转换成Spark streaming任务，而不是一次处理整个文件，尝试着微批次处理它，相比一次性整体做批量put而言，Spark streaming的方式可能会让快速地在HBase表中看到结果数据。

扫描的改进

尝试修改初始化scanner以只对某些行进行排序，或从你的表中只提取某些列，推送给Spark RDD的scan对象将决定你将在随后的调用中接收到什么数据。

第12章

用例：文档存储

最后的用例主要关注如何将HBase用为文件存储。这一用例是由一家大型保险公司实施，以下我们称其为“该公司”。该公司将收集的信息用于判别有问题的事故、支出和个人净资产。这个特定的用例包含了大规模数据管理和内容管理。

该公司需要构建一个文档存储，给众多的业务单元和顾客提供不同的文档服务。这种规模的公司会有成百上千的业务单元和数以万计的终端用户。这些不同的业务单元的数据会形成海量信息，有待收集、聚合和综合处理等，然后服务给内外的顾客。该公司需要具备在成千上万不同逻辑集合上服务上亿文档服务的能力。对于任何平台而言，这都是一个挑战，但对于HBase平台则不是。

利用HBase作为文件存储是一种相对较新的HBase的用例。在此之前，大于100KB的文档不推荐使用HBase。为了赢得生产部署，HBase与关系型数据库系统开展了竞争。在这一较量中，测试了读、写和文件大小等。作为现有的提供商，关系型数据库系统在面对HBase上也表现良好。HBase在写入较小文档（包含了从4KB到100 KB）时候速度略快。当然，现有的提供商在大文档上的表现优于HBase，在超过300MB文档上展示了两倍的写入速度。HBase在读出性能上也有良好的表现，在这一案例中，这一点对于终端用户非常重要。当读小文件时，HBase拥有四倍的速度，当读300MB以上的文件时，拥有3.5倍的速度。最后，HBase在测试900MB大小的文档时也表现良好，但生产部署环境不会有任何超过900MB的文件碎片。

这个用例在HBase早期版本中原本会出现问题。幸运的是，HBASE-11339中引入了MOB存储特性。刚开始HBase一步步解决100KB、1MB，以及更大的单元格。当使用这些较大的单元格，并且没有MOB功能可以用的时候，HBase会碰到所谓“写入

放大”的问题。当HBase 合并不得不一次又一次重写这些较大的单元格时候便会出现这种情况，可能导致刷新延迟、更新阻塞、磁盘I/O飙升和延迟大大增加。在将HBase用作记录系统，在Spark或者MR任务中更新大型数据集的批处理系统中可能没有问题，但是在SLA的实时系统中会比较糟糕。

MOB特性使得HBase可以容纳较大的单元格，官方推荐的是100KB到10MB，但是我们看到超过100MB的文档也能成功容纳。MOB是通过将MOB文件写入到一个专门region中解决这一问题的。MOB文件仍然是写入到WAL或者队列缓存中，以此支持普通的复制和快速的读取。但当刷新含有MOB文件的memstore时，仅将将一条引用写入到HFile中。真实的MOB文件被写入到离线的MOB region中，避免在较大文件合并时候，出现反复合并导致的写入放大。图12-1解释了使用MOB过程中的读路径。

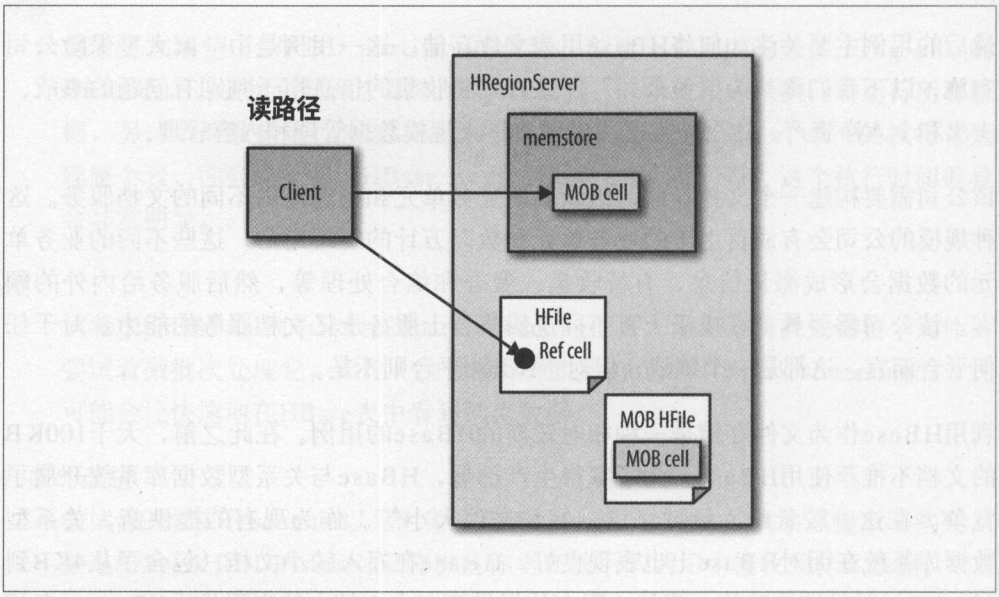


图12-1：理解MOB读路径

为了完成部署，该公司需要有在主集群上存储1PB数据，以及1 PB集群数据容灾恢复的能力，同时还要保持成本效益。保持成本效益意味着要在更大的垂直扩展性上降低集群总节点数。为达到这个目标，我们必须推动HBase超过已知的最好实践。我们让每个region保持在40GB，每个region服务器大概有150个regions。这样，光HBase便有大概6TB的原始数据（不包含用于合并的暂存空间或者Solr索引）。MOB的这一特性通过隔离大文件，使得我们能够更好的利用I/O系统。有了这个特

性，该公司能部署密集的节点，让每个节点提供24TB以上的存储容量。对于这个用例，我们将能专注于数据服务、数据摄取和数据清洗。

数据服务

我们将倒过来先介绍服务层，以便在了解怎么拆分数据前，更好地理解键的设计。第一步，客户端（或者终端用户）达到应用层去获取文档（见图12-2）。在这个例子中，客户端不会知道，当查询特定文档的时候，HBase的行键怎么呈现的。为了做到这一点，该公司使用Solr搜索引擎去查询特定的文档信息。搜索引擎中包含了需要被构建为HBase行键的文档的有关元数据。客户端发送他们搜索的词到Solr，然后基于搜索的结果，从HBase中请求一个或多个文档，搜索结果的信息包含了：

GUID

这是一个杂凑文档ID。

Partner ID

识别文档的起源点(例如，美国、加拿大、法国等)。

Version ID

对应搜索的文档版本号。

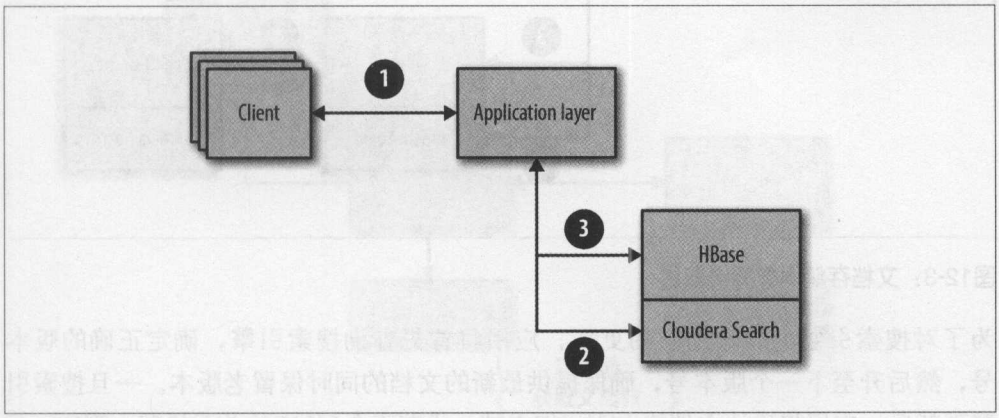


图12-2：文档存储的服务层

应用层从搜索引擎的检索来的信息中构建行键：

```
GUID+PartnerID+VersionID
```

当应用层从搜索引擎中构建了行键以后，然后将对HBase执行一个get操作。get操作针对整行，每一行代表完整的文档。文档以组块形式分入若干个单元格，应用层负责从这些单元格中重新构建一个文档。文档被重新构建以后，返回给终端用户更新，并重新写回到文档存储中。

数据摄取

数据摄取部分是非常有趣的，因为所有的文档大小都不同（见图12-3）。第一步，客户端更新或者创建一个文档到应用层，应用层获取文档，为将来的文档获取创造必要的元数据：

- GUID
- Partner ID
- Version ID

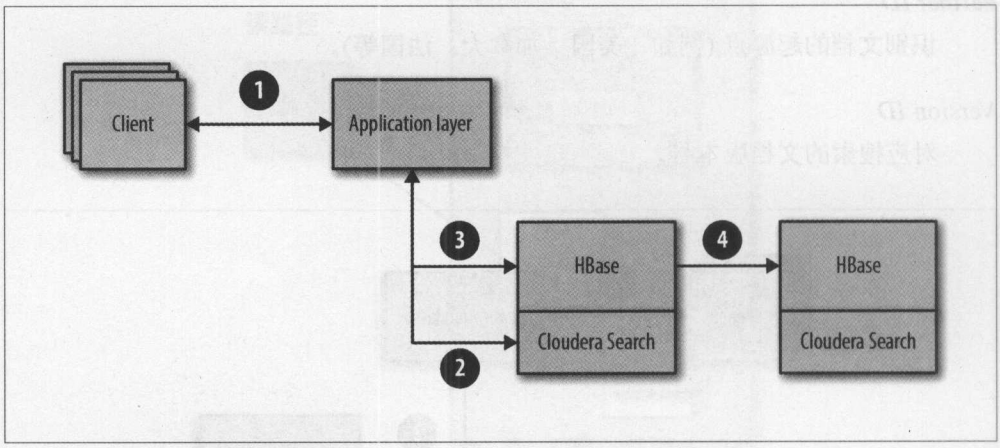


图12-3：文档存储的数据摄取层

为了对搜索引擎创建一个新的更新，应用将首先查询搜索引擎，确定正确的版本号，然后升至下一个版本号，确保提供最新的文档的同时保留老版本。一旦搜索引擎更新后，应用将确认文档的大小，把文档分成若干个50MB的单元格写入到HBase中。这个意味着250MB的文档将看起来如图12-4所示。

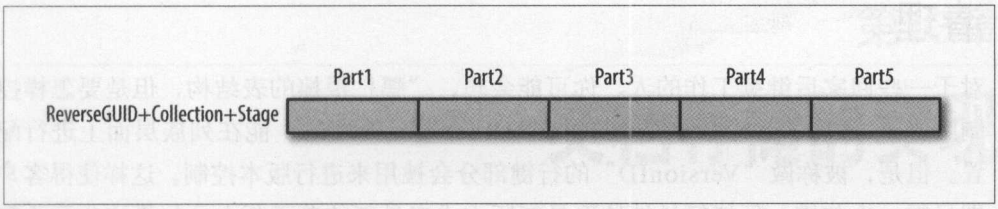


图12-4：分割大文档时候的单元格布局

最后HBase的schema当然和在评测中测试的不同。当文档如图12-4所示被分成数据块后，HBase拥有最好的性能。分块也有助于控制总memstore使用，因为我们可以一次刷新很多的数据块，而无需缓存整个大文档。一旦HBase应用将数据写入到搜索引擎和HBase中，文档便可以用在客户端进行检索了。

一旦数据写入到HBase中，Lily Indexer获得每一个文档的元数据，然后写入到Cloudera搜索引擎中。尽管搜索引擎在第4步中索引一些元数据，HBase也会复制数据到容灾恢复的集群中，该集群进而通过如图12-5所示的Lily Indexer写元数据到Cloudera搜索引擎。这样不用通过HBase资源扫描文档的总数量，能较快地获取文档数，实际上是一种非常聪明的方法。Cloudera搜索引擎能在这两个地方很快地获取到文档的总数，确保文档数量保持不变。

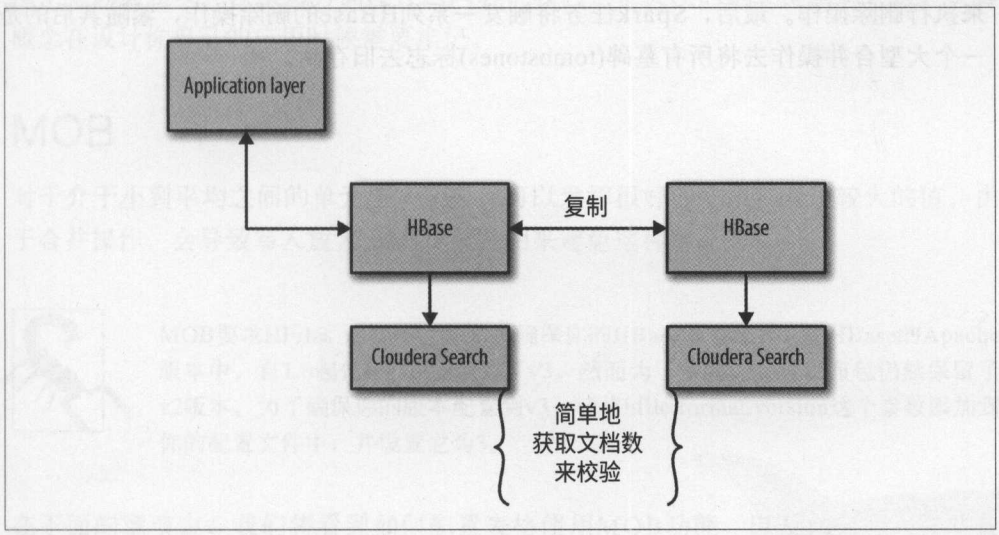


图12-5：容灾恢复的设计

清理

对于一些回家后继续工作的人，你可能会想，“嘿！很棒的表结构，但是要怎样控制版本?”通常，各个版本是由单个的HBase单元处理的，能在列族层面上进行配置。但是，被称做“VersionID”的行键部分会被用来进行版本控制。这样使得客户端只需一次扫描，便能轻松地拉取最新版本或者最新的若干版本。如果该公司希望保存所有单个文档的副本，那便还好，但是会像气球一样迅速失控，具体取决于文档的生成速率。严格的讲，想象下你反复修改了一个句子，急匆匆的保存文档，然后接着修改另一个句子，周而复始。为了解决这一问题，该公司编写了一个清理任务，遍历表格来删除文件的不需要的版本。

对于当前基于关系型数据库系统的解决方案，为了尽量减少昂贵的关系型数据的存储和包含成本，清理操作会不断执行。因为HBase是低成本的，存储算不上什么问题，清除操作可以不用那样频繁。为了做到这一点，该公司每天执行一个Spark任务。

Spark任务有两个关键的任务。第一，它收集搜索引擎中每一个文档的版本数量，这是消除不超过最大版本数的文档的一种简单方法 。在这个案例中，该公司保留了最后10个版本的文档。一旦文档列表被构建，Spark任务将利用响应去构建必要的行键来执行删除操作。最后，Spark任务将触发一系列HBase的删除操作，紧随其后的是一个大型合并操作去将所有墓碑(tombstones)标志去旧存新。

第13章

文档存储的实现

你可能已经猜到，这个用例将再次用到我们前面了解大部分（即便不是全部）大数据概念——复制备份、用做实时索引和搜索的Lily和Solr、支持快速处理的Spark、当然还有Java API。我们之前唯一没有了解的是MOB，因此在接下来的章节我们将关注这个方面。由于一致性也是这一用例的重要一方面，我们也将介绍这个内容。

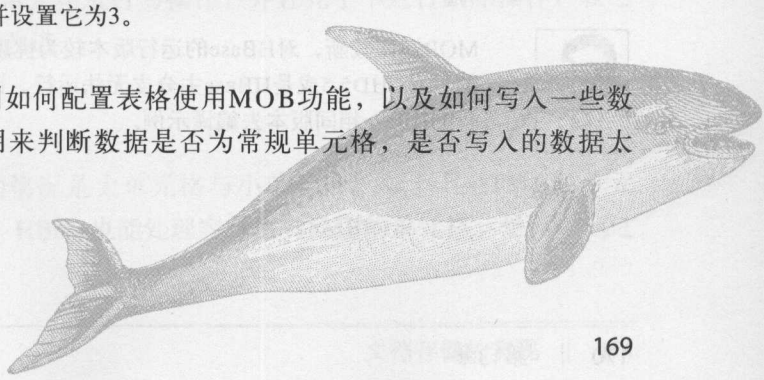
接下来你会发现，在实施方面没什么内容，我们主要关注的是关键概念，这些关键概念在设计你自己的应用时候需要牢记。

MOB

对于介于小到平均之间的单元值，HBase可以发挥很好的作用；对于较大的值，由于合并操作，会导致写入放大。MOB就是用来避免这种情况的。

MOB要求HFiles v3版本。你必须确保你的HBase版本配置。在HBase的Apache版本中，自1.0起便HFile配置到了v3。然而为了兼容，一些发布包仍然保留了v2版本。为了确保你的版本配置到v3，请将hfile.format.version这个参数添加到你的配置文件中，并设置它为3。

在下面的章节中，我们将看到如何配置表格使用MOB功能，以及如何写入一些数据？你需要确定一个临界点用来判断数据是否为常规单元格，是否写入的数据太大，需要移入MOB region中。



为了测试起见，我们将这个值设得非常小：

```
create 'mob_test', {NAME => 'f', IS_MOB => true, MOB_THRESHOLD => 104857}
```

MOB是作为HBASE-11339的一部分而实施的，已被纳入HBase2.0分支。不过还未被被移植到1.0分支。因此，如果你想要尝试MOB功能，必须使用2.0+以上的HBase版本，或者移植后的发行版。因为Cloudera QuickStart VM包含了该功能，我们在本章中用它做测试。

这一命令创建了一个称为mod_test的表，带有一个f列族，其中，超过几十兆的单元格将被认为是一个MOB。在客户端侧，不需要添加特殊的操作和参数。因此，put命令将把完整的单元格内存转移到Region Server中。一旦转移成功，Region Server会决定是否将文档存储为一个普通的单元格还是一个MOB region。

对于我们的mob_test表，从客户端的角度来下面的两个put非常类似，但以不同的方式存储在服务端：

```
byte[] rowKey = Bytes.toBytes("rowKey");
byte[] CF = Bytes.toBytes("f");
byte[] smallCellCQ = Bytes.toBytes("small");
byte[] bigCellCQ = Bytes.toBytes("big");
byte[] smallCellValue = new byte(1024);
byte[] bigCellValue = new byte(110000);
Put smallPut = new Put(rowKey);
smallPut.addColumn(CF, smallCellCQ, smallCellValue);
table.put (smallPut);

Put bigPut = new Put(rowKey);
bigPut.addColumn(CF, bigCellCQ, bigCellValue);
table.put (bigPut);
```

同一张表的相同行键的put操作都将进入同一个region，但是，等memstore刷新以后，较小的put将进入一个常规的HFile，然后较大的put将作为一个独立的MOB文件。

MOB还比较新，对HBase的运行版本较为挑剔。先前的示例是使用CDH5.7编译的，对CHD5.5或是HBase主分支无法运行。请确保你修改了POM文件，用你运行的HBase相同版本去编译示例。

存储

在第一部分中，我们了解了HBase怎样将数据存储到HFile中。每一张表被拆分成region，每一个region是一个HDFS上的目录，每一个列族是一个region目录的子目录。对于我们现在所看的表，当所有内容从内存刷新后，如果我们觉得没有使用任何专门的命名空间，下面的目录结构将被创建在HDFS上：

```
├── data
│   ├── default
│   │   └── mob_test
│   │   └── 4ef02a664043021a511f4b4585b8cba1
│   │   ├── f
│   │   │   └── adb1f3f22fa34dfd9b04aa273254c773
│   │   └── recovered.edits
│   │   └── 2.seqid
```

这看起来非常像任何普通表所呈现的情况。但是，我们现在在HBase目录下创建有一个MOB目录：

```
├── mobdir
│   └── data
│   └── default
│   └── mob_test
│   └── 0660f699de69e8d4c52e4a037e00a732
│   └── f
│   └── d41d8cd98f00b204e9800998ecf8427e20160328fa75df3f...0
```

正如你这里看到的，仍然有普通HFile文件包含了未被当成MOB过滤的小单元。但还有一个文件包含了MOB值。诚然，当HBase将文件合并到一起时，它要读出完整的HFile，产生一个新HFile包含有合并的内容。合并的目的是减少小文件的数量，创建更大的文件。因为MOB单元已经是比较大的文件了，将它们分开存储能够减少由合并引起的读写放大。每一次合并，仅MOB单元格引用会被读取或者重写。

MOB文件仍然有待合并。你可能已经删除了存储为MOB的值。有一个Shell命令可以做到，但是当请求的时候，HBase同样也会自动的触发合并操作。

如果你主要用HBase对较大的单元格进行写操作，并且几乎不进行删除操作，较之常规的格式，MOB会有较大的改进。

用法

此时，你可能想知道什么样的情况是大单元格与小单元格。对于上至1MB的单元格，HBase表现非常好。同样，HBase也能处理容量达到5MB的单元格。然而，超过

这个容量，常规的写路径会有性能上的挑战。另外一件你必须考虑的事是，较之常规单元格，你要将大单元格写操作多少次。MOB需要HBase做一些额外的工作。当你读取一个单元格时，我们首先需要去查找相关的HFile文件，然后再查找MOB文件。由于这一额外的调用，延迟将有轻微的影响。现在，如果你的MOB是50MB，相比传输数据到客户端的时间，HBase侧的额外读取可能不是很重要。HBase侧的额外读取在你的调用只会增加几毫秒，但是传输50MB的数据到你们的客户端可能远远不止这个时间，可能会掩盖掉额外读取多出来的几毫秒开销。

你需要问你自己的主要问题是，你要想以什么样的频率将大单元格插入HBase。如果你只打算偶尔为之，每行不超过一次，那么单元格可以比普通的大一些，比如到5MB，你可以使用常规的put API。这样你不用额外管理MOB文件，HBase会帮你处理。由于仅有数个较大的单元格，读写放大将会有限。但是，如果你打算每行都有多个较大的单元格，如果一些单元格超过5MB，或者大部分你写入的数据比较大，你可能需要考虑使用MOB。

记住，即使MOB会降低常规HFile的写放大程度，但如果你必须执行一些大单元格的大量输入和删除操作，你还是要对MOB文件进行大量的合并操作。

太大

正如我们刚刚看到的，常规的HBase的单元格偶尔增大到5MB很正常。MOB能使你轻松处理大至20MB的单元格。虽然不是很推荐，但有些人比较激进，对50MB的单元格也使用MOB。

现在，如果你的用例是只存储100MB或者更大的单元格，最好是另觅良策。

关于这一点，有两个最佳实践。首先，如果你的单元格超过100MB（而且你知道，它们会一直超过这个值），一种较好的方法是，把它们存为HDFS上的一个文件，然后将文件的引用存储到HBase中。记住，将数以百万计的文件存储在相同的目录中是不明智的，所以你应该根据你的用例设计一个目录结构（比如，你可以按月份、按日期或者按来源创造新文件夹）。

第二种方法即前面的章节中介绍的方法。如果单元格中只有少数较大，其余比较小，并且你事先不知道大小分布情况，可以考虑把单元格拆分成多个小的单元格。

然后，当写入数据的时候，检查它的大小，在必要的时候拆分它们，当读取数据的时候，执行相反的操作，然后合并成原来的数据。

这样做的最好办法是从客户端应用中提取。诚然，如果客户端自己拆分了内容，将会加大复杂性，可能会隐藏一些业务逻辑。然而，如果你扩展现有的HBase API，在写入HBase之前去解析Put操作，你就能顺手轻松地完成拆分操作。同样在Get操作时，获取HBase之前也要解析这些内容，合并它们到一个单独的对象中。

例13-1给出了如何实现的示范。为了简单地阐明原理，我们不扩展客户端的API内容。

例13-1：将一个HBase的单元格拆分成多块

```
public static void putBigCell (Table table, byte[] key, byte[] value)
            throws IOException {
  if (value.length <10 * MB) {
    // 如果值比较小，直接做为HBase一个单独的单元格，
    // 不需要拆分成多个单元格
    Put put = new Put(key).addColumn(columnFamily, columnQualifierOnePiece, value);
    table.put(put);
  } else {
    byte[] buffer = new byte[10*MB];
    int index = 0;
    int piece = 1;
    Put put = new Put(key);
    while (index <value.length) {
      int length = Math.min((value.length - index), buffer.length);
      byte[] columnQualifier = Bytes.toBytes(piece);
      System.arraycopy(value, index, buffer, 0, length);❶
      KeyValue kv = new KeyValue(key, 0, key.length,❷
                                 columnFamily, 0, columnFamily.length,
                                 columnQualifier, 0, columnQualifier.length,
                                 put.getTimeStamp(), KeyValue.Type.Put,
                                 buffer, 0, length);
      put.add(kv);
      index += length + 1;
      piece ++;
    }
    table.put(put);
  }
}
```

❶ 我们不想每次读一个新的分片就要创建一个新的10MB对象，所以我们重复使用同一个缓冲数组。

❷ HBase写对象不允许给出一个字节数组或者指定数据的长度，所以我们必须利用KeyValue对象。

确认你没有创建多个容量太大的行，HBase会验证你尝试存储的单元格的大小。如果你尝试用10MB的单元格，要确认*hbase.client.keyvalue.maxsize*没有被设为一个较小的值。这是一个相对比较不为人知的设置，当处理大单元格时，可能会导致很多头疼的问题。

数据一致性

让我们看一个小例子来真正的感受下什么是HBase的一致性。试想下，你正在将网站订单写入到HBase中，假设你用订单ID和接口ID构成的不同键逐条写入所有接口。订单中的每一行有一个不同的接口ID，同样，会有不同HBase行。大部分时候都会运行正常，会立即写入HBase。但如果是分布在两个不同的服务器上的两个不同的regions上会怎么样？如果第二个服务器恰好在存储接口之前出现故障了会怎么样？有些写入会成功，有些将不得不重试，直到region被重分配好，数据最后才会写入进去。当发生这种情况的时候，如果有个客户端应用试图读取完整的订单，它会只获得订单的一部分。如果我们用Lily给表中的数据建立索引，查询索引将只返回部分结果。

这就是一致性意识的重要性所在。

让我们看看在当前这个用例中是怎样运用的。

文档被应用接收，修改过的HBase客户端接口会把文档拆分成10MB的数据块，并把所有的这些数据块和元数据写入到HBase中。所有的数据块和元数据必须在一起存储。如果将这些数据块发送到不同的HBase行中，可能会花费大量的时间才能让每一部分都写入到HBase，这个时候，当有应用尝试读取的时候，可能只读到部分文件。对于所有数据块，重要的是单次写入，一次性读取，所以数据块必须被存在一行中。由于元数据用来建立索引，来于检索文档，所要也需要满足一致性准则，它和文件之间是紧密耦合在一起的。它们必须在同一时刻写进同一行。

在有的案例中，一致性不成问题，例如，你从一名用户那里收到一份推特列表，而这些微博之间没有什么关联。因此检索它们里面一个子集对你的结果可能不会有什么影响。

一致性在HBase中是非常重要的内容。并非在任何时候都需要在一个操作中保持一致性。但如果必须保持的话，要确保数据在一起，要么一起进入HBase，要么都不进入HBase。

远离交叉引用。让我们想象下，行A中存储了行键，引用值称为行B。当更新行B的值时，你还得更新行A的值。但是，它们可能分属于不同的region，更新操作的一致性是无法保证的。如果你读行A的时候需要获取行B的值，最好是使行B的键值保持在行A中，在这一基础上另行调用。这将增加一次额外的HBase调用，但是一旦出现一些故障，不得不重建完整的表的时候，将省去额外的负担。

进一步

如果你想扩展本章中前面的示例，基于我们在这章中讨论的内容，下方的清单将会提供一些选择供你尝试：

单个单元格

试出HBase的最大的容量。逐次尝试创建越来越大的数据块，而不是把单元格拆分成10MB的块，观察HBase的表现。尝试读写这些巨大的单元格，验证它们的内容。

删除的影响

为更好地理解怎样用MOB处理删除，更新该示例，生成一些较大的MOB单元格，将它们刷新至磁盘中，然后执行删除。然后看下面的文件系统的存储情况。最后，再次刷新表，执行合并操作，看情况如何。

Read读操作

本章阐述了怎样将大单元格拆分成小单元格，然后写入HBase中。但怎样写读路径，留给读者自行尝试。基于单元格拆分成片的方法，构建一个方法读取，并返回并重新创建初始文档。为便于你着手，你要找到最后一列，读取它并找出它的大小。文档的大小的计算公式为：(列的数量-1) × 10MB+最后一列的大小。举例而言，如果有三列，最后一列是5MB，那么文档的大小将是(3 - 1) × 10 MB + 5 MB，即25MB。

第三部分

疑难问题的定位和排错

当部署HBase用例的时候，可能出现很多的疑难问题。接下来的章节主要介绍了我们多年来在HBase上遇到的一些的问题。其中不少生产中的错误会使得恢复很棘手，所以建议你在部署HBase之前先阅读这一章节。

这个章节包含了我们遇到的典型问题，比如region和列族过多、热点问题、访问region超时，或者是最糟糕的情境：不得不从元数据或文件系统的崩溃中恢复过来。我们将研究典型原因和问题恢复的方法。我们还会重点介绍一些关于Java调优的最佳实践，使得HBase拥有更好的水平扩展性。

第14章

region过多

后果

region过多会处处影响你的HBase。

最常见的后果和 HFile合并相关，region之间会共享memstore的内存区域，因此，region越多，memstore刷新越小。当memstore满后，就不得不被刷新到硬盘，会创建一个数据存储在HDFS上的HFile。这也就意味着region越多，产生的HFiles越小。这个也迫使HBase执行大量的合并操作才能保持HFile的数量低至合理的数目。这些合并操作将会造成集群扰动过多，影响集群性能。当特定的操作被触发（比如自动刷新、被迫刷新和用户主动请求合并），HBase将会进行合并。当大量合并接踵而至，将会成为合并风暴。

合并是普通的HBase操作，小的合并操作没什么担心的。但是建议监控合并的数量，合并的队列不应该持续增长。峰值不是问题，但大部分时候要使键接近零。不停地合并会被认为工程设计糟糕或者集群大小有问题。如果region 服务器的合并队列始终不能降到零，或者不断增长，我们把这种情况称为持续合并。

如果region太多，有些操作会超时。分裂、刷新、批量加载和快照（当刷新的时候）等HBase命令会在每一个region上执行操作。举例而言，当flush<table>命令被触发时，对于memstore中等待进行写操作的<table>，HBase需要为其每个region的每个列族将单个HFile存入HDFS。region越多，在文件系统上创建的文件也会越

多。因为这些新文件的创建和数据的迁移都发生在同一时间，很有可能会压垮你的系统或者影响你的服务等级协议。如果完成操作的时间大于配置的超时时间，则会出现超时故障。

这里是集群上region太多时，你可能会面临的一些问题的简要总结：

- 快照超时。
- 合并风暴。
- 客户端超时（比如刷新）。
- 批量加载超时（可能会报RegionTooBusyException的异常）。

原因

region过多有多种根本原因，从配置错误到误操作不一而足，这些根本原因将在本节中做进一步的详细讨论：

- Region的最大值设置的太小。
- HBase升级的时候配置没有更新。
- 意外的配置设置。
- 分裂过多。
- 预分区不合理。

配置错误

在之前的HBase版本中，region大小默认为1GB。受硬件中默认内存大小不断增加以及HBase本身优化所赐，新版的HBase可以轻松地处理大小为10～40GB的Region。如果想详细了解如何用更多的内存运行HBase，可以参见第17章。当老版的HBase集群迁移到新版（0.98或者更高版本）时，通常会发生的错误是新机器的配置还在用老版本，这样导致：HBase原本来让region增长到10GB或者更大，但是仍然保持它们在1GB以下。即便集群不是从老版本迁移过来，配置中region大小的最大值仍有可能被错误地改成了较小的值，导致region数目过多。

误操作

同样，有多个手动操作可能会导致HBase的region过多。

分裂过度

首先是分裂功能的误用。HBase的“表详细信息”Web后台允许管理员将特定表的所有region分裂。只要region中的数据量满足手动分裂的最小尺寸，HBase就可以不断地将region分裂。这个命令选项有时候非常有用，但是一旦滥用，会造成某一表的region过多。举例而言，一个表有128个region，分布在一个四节点集群上，每一个RegionServer服务器平均拥有32个region，这是这种大小的集群能承受的正常负载，运用分裂选项三次，将产生1024个region，每一台平均256个region!

预分裂不合理

用HBase的预分裂功能同样可能造成region过多。预分裂能使负载分散到集群中的所有RegionServer上，有着非常重要的作用，但是使用它一定要事先深思熟虑。应该预先了解基本的HBase行键设计、写操作，以及正确将表预分裂所需的region数。预分裂不当，会造成region过多（成百上千个），但仅少数region能被用到。

解决方案

region过多的问题有不同的方法可以解决，具体取决于你所用的HBase版本。但无论是何种情境，最终的目标都是合并一些region，降低region总数，不同HBase版本之间的差别在于实现这个目标的方法。

0.98之前的版本

在0.98版本之前，合并region有两种主要的方法：拷贝一个表到一个新的预分裂的表，或者直接合并region。在0.98版本之前，region只能离线时候合并。

第一种方法是创建一个和已存在表的定义相同、但是表名不同的新表。将原来那个表的数据拷贝到新表中。完成后，丢开掉原来的表，以旧表表名重命名新表。这个方法需要用HBase的快照功能（HBase0.94.6版本以上）。同样你需要在HBase集群至上运行MapReduce 的拷贝表的任务，所以需要确保MapReduce框架是否可用或者能运行。最后，这个方法需要暂停数据写入，这个中断过程是短暂的，但是是整个过程重要的一部分。

因为这个方法要复制原表，会消耗和原表一样大的集群空间，所在开始前要确保有多余的空间可用。了解运行MapReduce对HBase集群的影响也很重要。

新表应该按照期望的新合并region之间的边界进行预分裂，如图14-2所示。一旦预分裂完成，便用CopyTable命令从原表拷贝数据到新表中。CopyTable工具以一个日期范围作为参数，然后将时间t0到t1的数据直接拷贝到目标表，拷入正确的region中。这使得你可以先传输旧数据，然后重新运行工具，再传输新数据。CopyTable同样允许你重命名或者丢弃不必要的列族，这个在表重设计的时候很有用。

如图14-1所示，如果你用修改过的时间戳进行puts和deletes操作，你应该避免使用这个方法，这是因为，当在两个CopyTable调用之间的来源侧出现任何合并的话，部分deletes和puts操作可能无法正确复制。修改内部HBase的时间戳绝不是什么好主意。在图14-1中，在t0到t1之间完成了CopyTable，把数据从一个表拷贝到另一个表。当在t1到当前时刻之间运行CopyTable拷贝数据时，一些印有旧时间戳的数据会插入到源表中。由于只有时间戳大于t1的数据才会在第二次运行CopyTable纳入考虑，t1之前新插入的数据将丢失掉（比如，这些数据不会被拷贝到新表，当源表被丢弃时，数据则将永久丢失）。

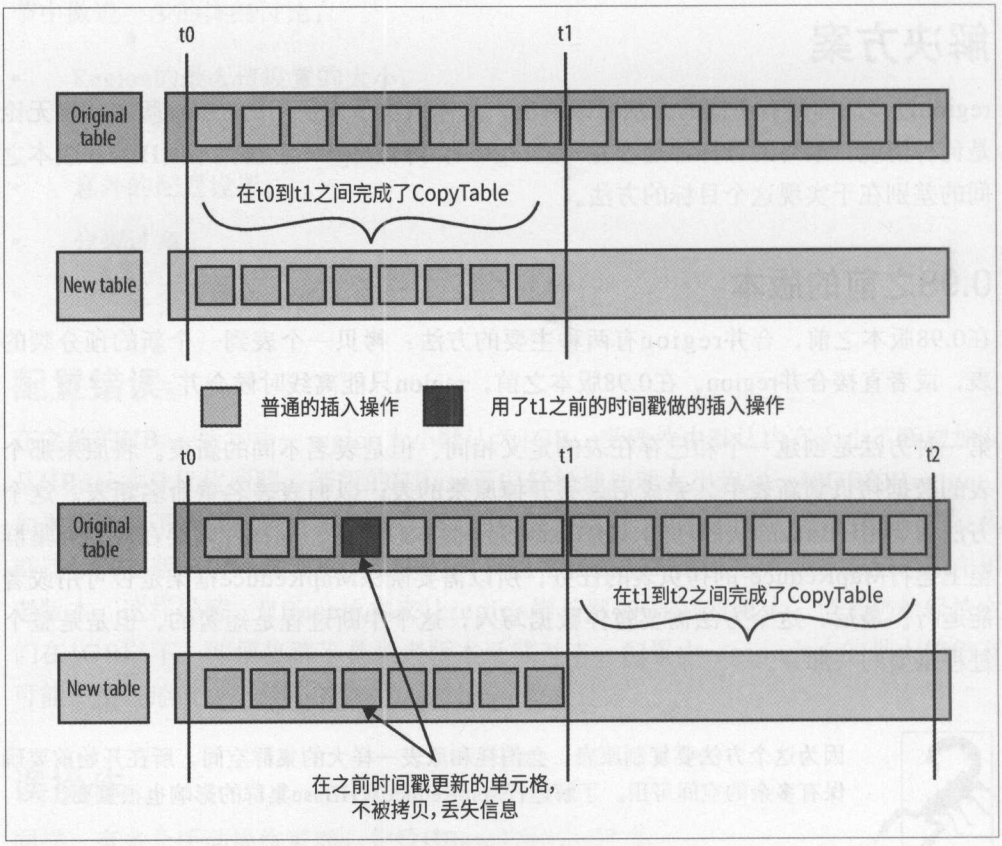


图14-1：在按单元格旧时间戳更新时存在的风险

图14-2的示例能更好的理解这个过程。假设你创建了一个应用，以事务ID作为行键将事务存储到一个表中。你最初的假设是，事务ID是一个随机产生的可读字符串，独一无二。基于这个假设，初始表可以被分成A到Z的26个region。但是当应用运行几周以后，你发现事务ID是十六进制的，因此，它是从0到F而不是从A到Z。因于这是一个十六进制的分布，大于F的分区便从来得不到利用，而第一个分区（存放了0到A的数据）获得远比其他分区更多的数据，并随着数据增长将分裂成诸多新region。这种情况下，我们把26个分区的数据仅移动到8个分区中。但是如果扩展此示例，你会看到成百上千的region最初是如何为了处理大的负载而创建的，以及同样的数据平衡问题是如何发生的。

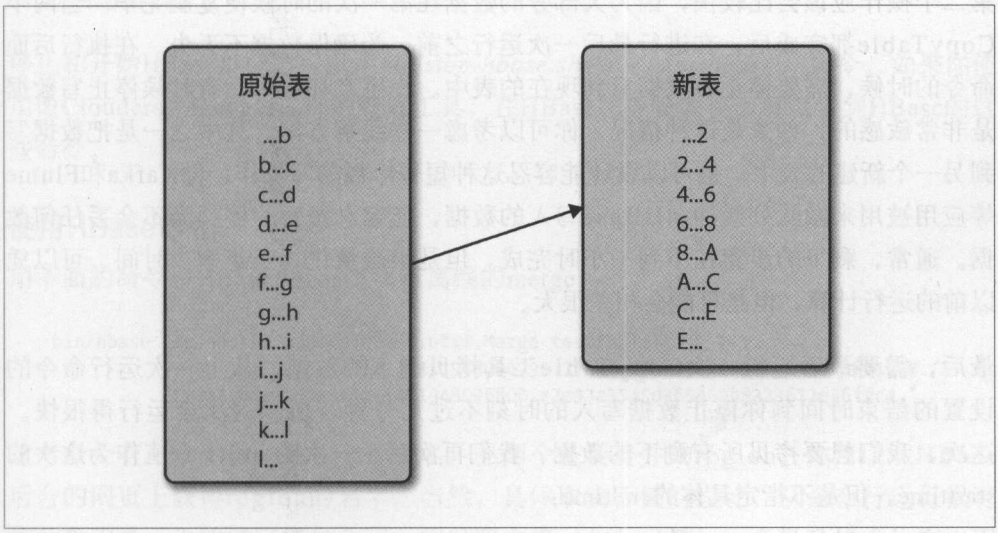


图14-2：region重设计

为了解决这个问题，我们建一个新表，按照十六进制预分裂。使用CopyTable命令，将数据从现有的表复制到新表，从t0到当前日期：

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --starttime=0 \
--endtime=1411264757000 --new.name=new_table previous_table
```

使用`starttime`和`endtime`参数的目的是确保我们拷贝的是源表中特定的可控区域段。因为这个操作可能会花一些时间，所在在进行下一步之前，我们要尽可能地拷贝足够多的数据，减少表的停机时间，但是我们也要确保能从一个特定的时间点重新启动拷贝操作。

拷贝操作可能花费几个小时到数天，具体取决于数据集的大小。当拷贝在运行的时候，普通的操作可以继续在这个集群或者初始表上执行。

一旦第一个CopyTable操作完成，新表除了你先前操作开始（在本例中，t=141126475700）到现在的数据没有外，其他数据都包含了。具体缺少多少数据，取决你第一个操作命令运行了多久，很有可能是不少的数据。有鉴于此，为了确保下一步命令执行的时间越短越好，我们要重新执行CopyTable命令。由于在先前的命令中使用的endtime参数是独一无二的，我们就用它作为当前的starttime：

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --starttime=1411264757000 \
--endtime=1411430414000 --new.name=new_table previous_table
```

第二个操作应该会比较快，因为大部分的数据在第一次的时候便复制完毕。当两个CopyTable都完成后，在进行最后一次运行之前，为确保数据不丢失，在执行后面命令的时候，需要停止将数据写到现在的表中。在生产环境中，有时候停止写数据是非常敏感的，如果是这种情况，你可以考虑一些缓解方案。其中之一是把数据写到另一个新建的表中；也可以设计能容忍这种短暂停顿的写操作，像Kafka和Flume等应用被用来做队列缓冲向HBase写入的数据，能容忍短暂的停顿而不会丢任何数据。通常，剩下的步骤在不到一小时完成，但是很难量化真正花多少时间。可以凭以前的运行计算，但是可能会相差很大。

最后，需要最后运行一次CopyTable工具拷贝剩下的数据。从上一次运行命令的设置的结束时间到你停止数据写入的时刻不过几分钟，所以这次会运行得很快。这次，我们想要拷贝所有剩下的数据，我们再次用上一次的endtime值作为这次的starttime，但是不指定具体的endtime：

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --starttime=1411264757000 \
--new.name=new_table previous_table
```

这个可能运行几秒钟到几分钟。在这个操作结束时，我们完整的复制了源表的数据到新分区的新表中。最后一步是运行几条快照命令重命名新建的表：

```
disable 'events_table'
drop 'events_table'
disable 'new_events_table'
snapshot 'new_events_table', 'snap1'
clone_snapshot 'snap1', 'events_table'
delete_snapshot 'snap1'
drop 'new_events_table'
```

完成这些操作后，表中包含的数据相同、但是少了不少region。

离线的merge

减少regions数量的第二种选择是执行离线merge。为了能够执行后面的步骤，需要运行0.94.13或者更高版本的HBase。0.94.13之前的版本进行离线merge可能会失败或者在你系统中留下一堆损坏的文件（参考HBase-9504）。离线merge受到的主要约束是它们需要集群的总共停机时间。

执行离线的merge有以下几步（稍后我们会详细研究）：

1. 停止集群。
2. 执行一个或者多个merge操作。
3. 启动集群。

停止和开始HBase集群，可以用*bin/stop-hbase.sh*和*bin/starthbase.sh*命令。如果你使用像Cloudera Manager这样的管理工具，在HBase服务管理页面可以看到HBase的这些命令。

使用HBase命令

用下面的命令中HBase Merge类执行离线的merge：

```
bin/hbase org.apache.hadoop.hbase.util.Merge testtable \
                    "testtable,,1411948096799.77873c05283fe40822ba69a30b601959."\
          "testtable,11111111,1411948096800.e7e93a3545d36546ab8323622e56fdc4."
```

这个命令有三个参数：第一个是表名，另两个是要合并的region。你可以从HBase后台的网页上获得region的名字。当然，具体取决于键的设计，从命令行运行很可能非常困难。你的键可能包含了一些保留字符（如$、^等），会导致格式化有点困难。如果是这种情况，则不能格式化命令，那么使用下文介绍的Java API方法可能比较明知。一旦执行命令，HBase会首先测试集群是否停止了，然后会执行多个命令去合并这两个region的内容到一个region中。这个过程同样会创建一具拥有新的边界的region，创建相关的目录结构，将内容移至新的region中，更新HBase .META.信息。操作的持续时间取决于你合并的region大小。

使用Java API

使用Java API可以执行同样的操作，我们充分利用HBase的merge工具类，在给出正确的参数时候调用它。这个示例专门针对HBase 0.94 和 0.96，不能使用HBase后面的版本编译。这里省略了部分代码，仅在例14-1中提供了一小段代码。

例14-1：HBase 0.94中Java的merge过程

```
public int mergeRegion (Configuration config, String tableName,
                        String region1, String region2) {
  String[] args = {tableName, region1, region2};
  int status = -1;
  try {
    status = ToolRunner.run(config, \
    new org.apache.hadoop.hbase.util.Merge(), args);
  } catch (Exception e) {
    e.printStackTrace();
  }
  return status;
}
```

这个方法通过直接传递表名和region名称，使得你可以绕开前面命令行解析问题。这种方法在HBase1.0中仍然可以用，但是因为在线merge也可以运用，所在在新版本中不推荐使用这种方法。

这个方法需要运行Zookeeper实例，在HBase 0.94的独立服务器上无法测试离线merge操作。你只能在伪分布式或分布式模式下运行这个方法。

自0.98版本起

HBase-7403自0.98版本起引入了在线合并功能。在线merge无需关闭HBase集群或者disable目标表便可将两个region合并。这是merge过程的极大改进。假设拥有一个足够高版本的HBase，这会是目前解决region过多问题的首选方法。

使用HBase shell

使用HBase shell，你需要用到`merge_region`命令。这个命令只需要两个以region编码的名称作为参数:

```
merge_region 'ENCODED_REGIONNAME', 'ENCODED_REGIONNAME'
```

以region编码的名称是整个region名字的最后标记，在region `testtable,22222222,1411566070443.aaa5bdc05b29931e1c9e9a2ece617f30.`中，“`testtable`”代表表名，“`22222222`”代表start key，“`1411566070443`”代表timestamp，最后的“`aaa5bdc05b29931e1c9e9a2ece617f30`”代表编码名称（注意,最后一个点是编码名称的一部分,有时是必需的,但对merge命令不需要）。

命令调用和输出类似这样：

```
hbase(main):004:0> merge_region 'cec1ed0e20002c924069f9657925341e',\
                                '1d61869389ae461a7971a71208a8dbe5'
0 row(s) in 0.0140 seconds
```

操作结束后，可以在HBase web UI上验证region是否合并成功、新的region是创建成功并取而代之，包括初始两个region的边界在内。

使用Java API

HBase Java API有同样的merge操作命令可用。在例14-2中，查询了一个表，获取了region列表，分别将region两两合并。执行到最后，原始的表的region数量减至一半。

例 14-2：在线Java合并

```
Configuration conf = HBaseConfiguration.create();
Connection connection = ConnectionFactory.createConnection(conf);
HBaseAdmin admin = (HBaseAdmin)connection.getAdmin();
List<HRegionInfo>regions = admin.getTableRegions(TableName.valueOf("t1")); ❶
LOG.info("testtable contains "+ regions.size() + " regions.");
for (int index = 0; index <regions.size() / 2; index++) {
  HRegionInfo region1 = regions.get(index*2);
  HRegionInfo region2 = regions.get(index*2+1);
  LOG.info("Merging regions "+ region1 + " and "+ region2);
  admin.mergeRegions(region1.getEncodedNameAsBytes(),
                     region2.getEncodedNameAsBytes(), false); ❷
}
admin.close();
```

❶ 从特定的表中获取现有region。

❷ 执行merge操作。

在这一示例中，表有9个region，输出如下（为适应页面的宽度，一些输出已截断）：

```
2014-09-24 16:38:59,686 INFO [main] ch18.Merge: testtable contains 9 regions.
2014-09-24 16:38:59,686 INFO [main] ch18.Merge: Merging regions ... and ...
2014-09-24 16:38:59,710 INFO [main] ch18.Merge: Merging regions ... and ...
2014-09-24 16:38:59,711 INFO [main] ch18.Merge: Merging regions ... and ...
2014-09-24 16:38:59,713 INFO [main] ch18.Merge: Merging regions ... and ...
```

每一个region将在输出中打印下面的信息：

- The encoded region name in the form of a 32-character long hexadecimal string: `4b673c906173cd99afbbe03ea3dceb15`

- The region name formed by the table name, the start key, the timestamp, and the encoded name, comma separated:
  testtable,aaaaaaa8,1411591132411,4b673c906173cd99afbbe03ea3dceb15
- The start key: `aaaaaaa8`
- The end key: `c71c71c4`

在这个示例表中，键是基于string的，可以被HBase打印。但是当你的键含有不可打印的字符时，HBase会用十六进制将它们格式化。同时包含可打印和不可打印字符的键，例如字节数组`[42, 73, 194]`，将打印成`*I/xC2`。

防范

有多个方法可以防止HBase集群面临关于region过多的问题。

记住，你不仅要考虑region的数量，也要考虑列族的数量。诚然，每一个region中的每一个列族，HBase都会跟踪它的内存和`hbase:meta`表。因此，400个拥有两个列族的region，不如800个拥有一个列族的region。表中列族过多同样会造成一些问题，我们将在15章探讨。

region的大小

首先，你要确保最大的文件大小设置为至少10GB。有些region小于10GB没问题，但是要确保，在必要的情况下，region能达到10GB。本章节稍后会讨论预分裂。推荐使用类似Hannibal的可视化工具监控region的大小。留意增长速度比较快的region，并伺机在非高峰使用时将它们分裂。图14-3描述了Hannibal怎么帮助你查看每一个regions的大小。

HBase默认的split策略（`org.apache.hadoop.hbase.regionserver.IncreasingToUpperBoundRegionSplitPolicy`）确保region绝对不会超过配置的最大值（`hbase.hregion.max.filesize`）。但是，当集群负载比较小的时候（少于100个region），可能会创建很多较小的region。为了确保你的region能增长至正常的大小，要验证`hbase.hregion.max.filesize`属性值是滞设置成了至少10GB。 这个属性值以字节表示，即是10737418240。当然，常见的做法是设置将最大文件大小设为更大的100GB。当使用较大的region大小时，需要手动管理split。确保split的时间合适，对HBase集群操作和SLA影响最小。

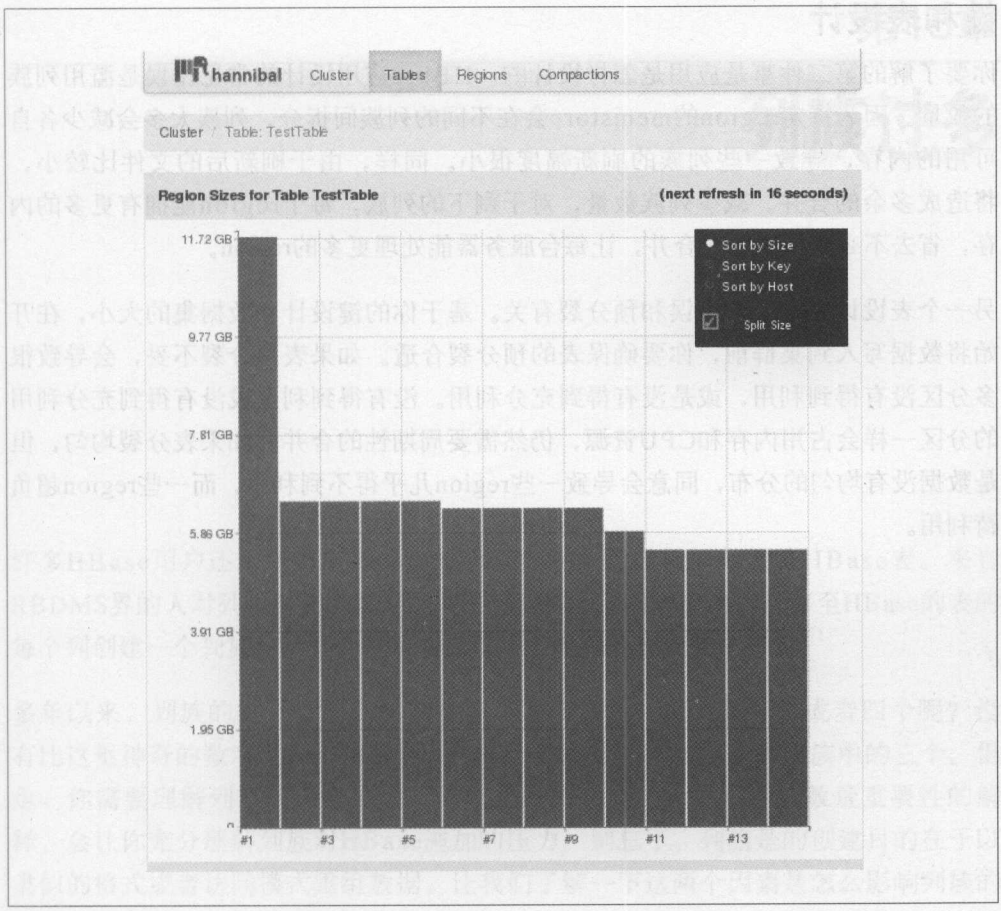


图 14-3：由Hannibal提供的region大小的视觉概览

不推荐禁用split，或者将split大小设置为200GB这样大的值。如果容量出现峰值，或者有未清理的记录，或者管理员休假了，region分分钟会变得笨重。如果region没有split，一直持续增长，很可能导致RegionServer崩溃。在出现故障之后，服务器上的所有region将被分配到另一个服务器上，可能由于大小问题而导致崩溃的级联效应。有问题的region从RegionServer重新分配到另一个RegionServer，每次会导致目标主机出现故障。如果不及早发现这个问题，整个集群可能会瘫痪。

即便你预计表很小，也最好做预分裂，扩大在RegionServers上的分布，或是手动触发split确保你的region跟RegionServer一样多。但是，由于HBase在较大的region上表现更好，所以要确保各个region的大小至少在1GB。

键和表设计

你要了解的第二件事是应用是怎样设计的。HBase应用设计的常见错误是滥用列族的数量。因为特定region的memstore会在不同的列族间拆分，列族太多会减少各自可用的内存，导致一些列族的刷新幅度很小。同样，由于刷新后的文件比较小，将造成多余的合并。减少列族数量，对于剩下的列族，每个region能拥有更多的内存，省去不必要的刷新和合并，让每台服务器能处理更多的region。

另一个表设计中常见的错误和预分裂有关。基于你的键设计和数据集的大小，在开始将数据写入到集群前，你要确保表的预分裂合适。如果表预分裂不妥，会导致很多分区没有得到利用，或是没有得到充分利用。没有得到利用或没有得到充分利用的分区一样会占用内存和CPU资源，仍然需要周期性的合并。如果表分裂均匀，但是数据没有均匀的分布，同意会导致一些region几乎得不到利用，而一些region超负荷利用。

第15章

列族过多

许多HBase用户还未了解所有的HBase功能和特性前就开始设计HBase表。来自RBDMS界的人对列族和列限定符之间的差异并不了解，会为打算移至HBase的表的每个列创建一个列族。结果便是会看到很多表的设计都列族过多。

多年以来，列族的数量建议在控制在三个以下。为什么不是二个或者四个呢？没有比这更神奇的数字了。严格来讲，HBase能应对的不止四个的列族中的三个。但是，你需要理解列族的工作原理才能充分利用它们。此处对列族数量重要性的解释，会让你充分理解列族对HBase施加的压力。别忘了，列族是的创建目的在于以类似的格式或者访问模式重组数据。让我们了解一下这两个因素是怎么影响列族的数量的：

关于格式

如果你要存储较大的文本数据，你很可能会打算压缩这个列族。但是如果你想要在同一行存储图片，那么你很可能不打算压缩列族，因为它将占用CPU周期，不节省任何空间，因此会对性能产生负面影响。所以将列族分散更为合理。

关于访问模式

介绍访问模式的最好的方法就是以真实的例子作为说明。假设有个表存储了客户信息，一些大的列存储客户的元数据，包含了很多元数据，大概几千字节的大小，而另一列存储用户单击网站某个页面的次数。元数据列基本从来不改变，然而存储访问次数的列每天会更新好几次。随着时间流逝，由于访问次数的更新操作，整个memstore将会被刷新至硬盘。这个操作将创建通常仅包含

计数操作的文件，然后在某个时候会被合并。当合并完成的时候，很可能会选择包含客户元数据的HFiles。这些合并操作会重写这些客户元数据大单元格，也会重写较小的计数文件。其结果是，大部分I/O操作被浪费在重写这些变动很少甚至没有变动的文件，仅仅为了更新和合并一些计数文件，这对I/O造成很大的开销。HBase在列族层面上触发合并。通过将客户元数据和客户访问计数分散至两个不同的列族上，可以避免了不必要的静态信息重写。最终降低了RegionServer的I/O负载，因此而提高了整体的应用性能。

那么，多少列族才算是太多？我们不能给你一个万能的数字。如果从格式或者访问模式上拆开列族是合理的，那么就分开。但是如果你几乎以相同的方法读写它们，数据几乎也是相同的格式，那就让他们留在同一个列族中。

后果

滥用列族将影响你的应用的性能，会处处影响HBase的反应。由于出现超时错误或者RegionServer瘫痪等，很可能影响HBase的稳定性，具体取决于你给HBase施加的压力。

内存

列族过多首先影响的是内存。HBase在各个region间共用了memstore（写缓存）。由于每个region允许最大的缓存配置大小为128MB，所以这个内存段必须被同一个region的列族共享。因此，列族越多，每一个列族在memstore中可用的内存越小。一个列族的内存区域满了后，那个region中所有其他的列族被会被迫刷新到硬盘，即便它们的数据相对较少。这将给内存造成很大的压力，会不断创建很多对象和小文件，由于这些小文件将不得不合并在一起，还会给硬盘造成一些压力。

HBase-3149和HBase-10201经过改进，只会刷新内存区域满的列族，而非所有列族。但是在HBase 1.0上这个功能还不可用，一旦这个功能可用，列族过多对内存的影响将大幅的减小，对合并的影响也会减小。

合并

列族的数量将会影响HBase在刷新过程中创建的store file的数量，以及后续必须执行的合并数量。如果一个表有8个列族，并且region的128MB memstore也满了，8个列族的数据将被刷新到分开的文件中。随着时间的推移，将会发生更多的刷新操

作。当一个列族中存在超过三个store file， HBase会让这些文件合并。如果一个表只有一个列族，便有一组文件需要进行合并。8个列族的表，有8组文件需要合并，影响Region服务器和HDFS的资源。列族设置较少，使得每一个列族有一个较大的memstore，这样需要被刷新的store file则会较少，更重要的是，合并操作也会减少，降低了HDFS上I/O的操作。当一个表需要刷新（例如在快照之前，或者管理员在Shell上直接触发刷新），所有这些memstores将刷新至硬盘。具体取决于前面的操作，这样做可能会使得HBase触发另一个合并，为许多（即便不是全部）regions和列族启动更多的合并。列族越多，加入队列的合并则越多，HBase和HDFS承受的压力越大。

分裂

HBase将列族中的数据存储至独立的目录和文件中。当其中的一个region大于配置的region大小时，将触发分裂操作。分裂将影响region中所有的列族，而非仅仅影响那些数据增长到最大值的列族。其结果是，如果一些列族非常大，而其他的则非常小，最后可能列族仅包含了区区一些单元。RegionServer会按region和列族分配资源，如内存和CPU线程等。极少数region和列族将对这些资源实施额外的压力。HBase 主机需要管理HDFS系统上hbase:meta表中更多的接口。HDFS上的DataNode和NameNode节点同样需要为这些较小的列族文件管理I/O操作。如果你希望部分列族比其他列族包含更多的数据，你可能需要把这些数据分到其他的表中，使得文件变少但变大。

原因、解决方案和预防

造成列族过多的原因始终离不开schema设计。HBase不会替你创建过多的列族，也不会有自动分裂操作造成列族过多。因此，为了防止这个问题，你在设计schema之前，需要仔细的考虑HBase的原理，确定合适的列族数量。

对于列族过多的问题有一些解决方案。了解列族中的数据和访问模式是非常重要的。有时候，如果数据被复制/反规格化，并且在另一个表中可用，那么不需要列族，尽管丢弃。有时候列族在存在是仰仗了访问模式（如列族的归纳或汇总）。在这种情况下，有可能列族可以从另一个表分享，移至自己的表中；还有的时候，单独列族中的数据可与一个较大的列族合并。

后面几节中的所有操作都能用Java API实现，但是也可以通过HBase Shell或命令行轻松完成。由于Java API对这些操作不会带来实质性的好处，我们对它没有正式记录。

删除列族

如果你认为不需要某个列族，就从表的META信息中删除它。下面的方法是从“sensors”表中删除“picture”列族：

```
alter 'sensors', NAME => 'picture', METHOD => 'delete'
```

这一操作可能需要一些时间执行，因为要逐个在所有region中运用。最后，HDFS中相关的文件将被删除，hbase:meta表将会更新，以便体现最新的修改。

合并列族

由于原始schema设计中的瑕疵，或者原用例范围的变化，你可能将数据分到两个不同的列族了，但是现在想合并回到一个列族中。正如我们在第14章讨论的解决方案，CopyTable能使你将数据从一个表拷贝到另一个表。在当前的情况下，CopyTable可以帮到我们。方法便是CopyTable会请求一个来源表和一个目标表，然后这两个表不一定需要不同。同样，CopyTable能使我们修改列族的名字(见图15-1)。

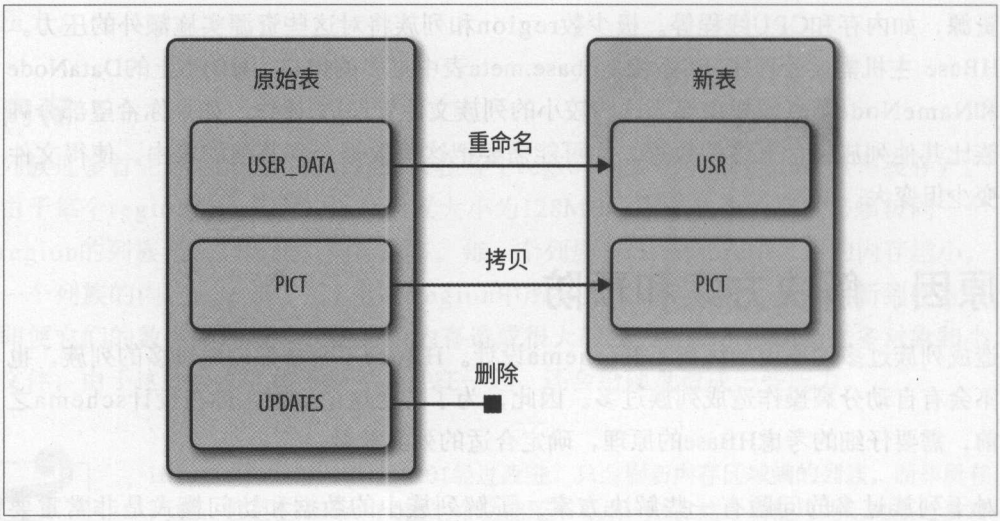


图15-1：CopyTable列族操作

CopyTable会在你想要读取的数据上运行一个MapReduce任务，然后基于你的请求提交写操作。如果对于某个被称为“customer”的特定表，你想将 “address”列族的数据移至“profile”的列族中，你只需要将输入和输出设置为“customer”表，设置输入列族为“address”，输出的列族为“profile”。通过运行如下的命令可以完成这个操作：

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --new.name=customer \
                                        --families=address:profile customer
```

MapReduce任务运行完后，所有在“address”列族中的数据都将出现在“profile”列族中。你现在可以通过先前代码片段中的alter命令删除“address”列族。

在使用CopyTable方法合并列族时有几个注意事项。目标列族的数据可能会被覆盖。如果在来源和目标列族中存在相同的行限定符或列限定符，目标列族的数据将会被覆盖。当拷贝列族时， HDFS要有足够的空间承受临时的拷贝。在开始这个操作之前，要预估需要的额外的空间。如果你在活动的生产表上运行这个命令时，要确保在原始列族上的任何更新同样被更新至目标表。如果你对put、delete使用的是现成的或自定义的时间戳，要避免在活动的表上使用这个方法，否则会造成不可预料的结果。

你也可以将多个列族合并回一个列族中，只需要用逗号分隔符分别指明：

```
--families=address:profile,phone:profile,status:profile
```

将一个列族分离到一个新表中

出于各种原因，可能需要把数据分到不同的表中。也许某个表的列族之间的操作不需要原子性，或者列族之间数据大小和访问模式有明显的差异。也许设置不同的表更合逻辑。我们再次利用CopyTable执行这个操作。

下面的命令是拷贝“customer”表的“picture”列族的数据到“map”表的同一个列族中：

```
hbase org.apache.hadoop.hbase.mapreduce.CopyTable --families=picture \
                        --new.name=map customer
```

再次声明，只要你想将多个列族移至同一个目标表中，你可以用逗号分隔符指明各个列族。

在你启动MapReduce任务之前，要确定目标表和目标列族都存在。

前段规定的所有警告也适用当前情况。

第16章

热点

正如前面所讨论的，为保持并行的特性，HBase利用了分布在node上的RegionServer中包括的region。在HBase中，所有的读写请求应均匀的分布在RegionServer上的所有region上。当单个RegionServer上的某个region收到大部分或者所有的读写请求时，会发生热点现象。

后果

HBase基于行键处理读写请求。行键是一种工具，以供HBase一视同仁地利用所有region。当发生热点现象的时候，处理所有的请求的RegionServer会变得不堪重负，而其他region服务相当空闲。图16-1展示了成为热点的region。单个RegionServer上的负载越高，则I/O密集、受到阻塞的待执行进程越多（例如，合并、垃圾回收、region的分裂操作等）。热点会导致延迟加重，在客户端超时，或错过SLA。

原因

导致热点的主要原因通常是行键的设计的问题。在下面的章节中，我们将研究热点一些最常见的原因，包括行键单调递增、行键分布不当、引用表极小，以及应用问题等。

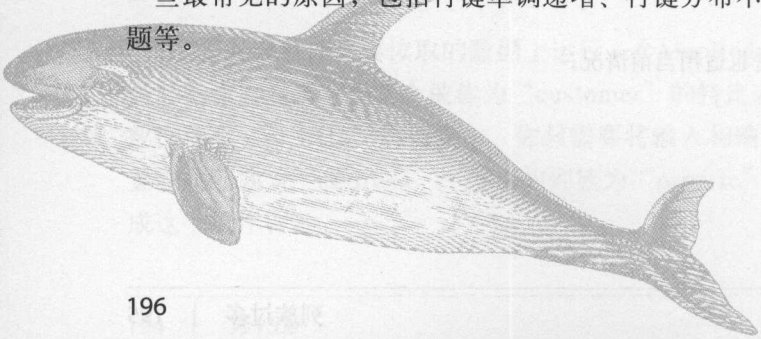


Table Regions

Name	Region Server	Start Key	End Key	Locality	Requests
TestTable,,1427677218929.c808f8d56f78c3c41fb9617c95c5d8fc.	t430s:16201		010094234	0.0	2209665
TestTable,010094234,1427677218929.979d5f6ed7856d60cf67973ae83747d3.	t430s:16201	010094234	020223584	0.0	667
TestTable,020223584,1427677269908.f9d413e1a9df96f0eccb4f24428ef71e.	t430s:16201	020223584	030929104	0.0	686
TestTable,030929104,1427677269908.a8c98da26de9c3138a00e1b2876cdf0b.	t430s:16201	030929104	050013975	0.0	1238
TestTable,050013975,1427677280538.6c053ebc240abea265ea800e09c82c0e.	t430s:16201	050013975	06007249	0.0	588
TestTable,06007249,1427677280538.5ba5a5cfb4660c2717ecf49cf5880098.	t430s:16201	06007249	07015627	0.0	630
TestTable,07015627,1427677283513.efdb955f394be47dac15bb0745631d13.	t430s:16201	07015627	080419479	0.0	655
TestTable,080419479,1427677283513.4a1ea52a4e57617067bbc879b11f8a25.	t430s:16201	080419479		0.0	1238

图 16-1：存在热点的Region

单调递增的行键

单调递增的行键指的是那些仅最后几个比特或者字节慢慢增长的行健。这意味着，写入HBase或从HBase读取的新行键大部分与前面读写的行键非常类似。当将时间戳用作键的时候，会出现最常见的单调递增行键现象。当时间戳被用作行键时，行键从最初写入开始慢慢增长。让我们看一个快速增长的示例：在HBase中，键按照字典序的顺序被存储。行键将更新如下：

```
1424362829
1424362830
1424362831
1424362832
...
1424362900
1424362901
1424362902
1424362903
1424362904
```

如果是写请求，前述各个更新都将进入同一个region，直到它到达下一个region的行健或者配置的region的最大值。在这个时候，region将会分裂，我们将以递增的方式更新下一个region。请注意，大部分的写操作是针对同一个region，因此会对单个RegionServer造成负载过大。不幸的是，在部署完后发现这个问题时，已经无法避免热点问题了。避免这种问题的唯一方法是做出合理的行键设计。

行键分布不合理

正如之前提到的，行键设计非常重要，因为它不仅将影响应用的扩展性，而且影响它的性能。然而，在用例中第一次迭代schema设计时，行键设计不佳并不少见，从而导致分布不合理。当schema设计缺乏信息的时候，执行应用产生问题的时候，都有可能发生。举个行键分布不合理的例子，你希望源数据发送给你行键，数字分布在“0”到“9”之间。但是你收到的键，在你期待的“0”到“9”之前总是有个前缀值。在这种情况下，应用期待收到“1977”和“2011”，但是实际收到“01977”和“02001”。在这个示例中，如果表合理地预分裂成10个regions（上至“1”、“1”到“2”、“2”到“3”等），那么你所有收到并存储在HBase的值，都将被写到第一个region中（上至“1”），而其他的region则空无所得。在这种情况下，即便我们对设计schema有着正确想法，数据也不会完全充分分布。通过进行合适的测试可以发现这个问题，但是如果在应用部署之后才发现，也不要绝望，局面是可以挽回的。建议将热点分区分裂为多个分区。这样能回到预期的分布情形。在这个示例中，你要将第一个region分裂成10个region，并考虑到前缀的0。新的region区域分布则为“00”、“00”到“01”、“01”到“02”、如此类推。如果键的前缀一直是“0”，并可以通过“10”region逐个合并，那么“1”以后的其他region将不会用。

引用表较小

这一常见的热点问题是指使用较小的引用表（通常是在HBase中聚合）造成的瓶颈。例如，有两个引用表，由定义邮政编码和城市名字的单个region组成。在这种情况下，我们在一个十亿行的订单表上执行一个MapReduce任务。所有产生的映射器将查询这两个表进行查找。其结果是，集群中所有其他服务器都来调用拥有这两个引用分区的两个RegionServer，使之不堪重负。如果足够倒霉，这两个region是同一个RegionServer伺服的。这种关联/瓶颈将增加延迟，造成延误，由于超时而引起任务失败。好消息在于，可以通过多个方法避免这种情况。首选的简单方法是将引用表预分裂，目标是让分区数和RegionServer数尽量一样多。如果表很小，或者在当前时刻，HBase上有太多的RegionServer，引用表预分裂不是明智之选。

另一个方法是在执行聚合前把表分布到所有节点上。这些数据的分布通过使用MapReduce分布式缓存机制完成。在任何任务执行之前，分布式缓存框架会将数据拷贝从属节点。由于在每个任务中，文件只拷贝一次，所以这种方法还是高效的。

当数据拷贝到每个服务器后，如果数据足够小，可以放进内存，那就能从MapReduce代码的setup函数加载这些数据，这些数据便不用从硬盘查找，可以直接从内存中查找。

这样即能提高性能，也能解决热点问题。

如果MapReduce也更新或者写入引用表，则分布式缓存不是一个明智选择。诚然，一旦开始拷贝一个表，就意味着任务运行期间的内容是固定的。如果在你的用例中要更新引用表，则最明智的选择是将引用表预分裂，确保表在RegionServer上均匀分布。

应用问题

最后一个处理region热点的示例和应用设计和执行的问题相关。当一个region存在热点现象时，了解根本原因是非常重要的。为了做到这点，我们需要确定有问题的调用来自哪里。如果数据合理地分布到表中，热点可能来自一个总是写入同一个region的bug。错误地添加了前缀、两倍或三倍重复写入，或者转换数据类型时候可能的潜在错误，都可以表现为应用程序问题。想象一下，系统原本要收到一个四字节的整数值，但是后端代码将它转换成高字节值，高字节值的四个初始字节可能会合理地分布在整个数据范围上，但是，增加四个空字节到这个高字节值上，将创建一个不会递增的前缀，这个前缀由四个空字节[0x00, 0x00, 0x00, 0x00]构成，都落到相同的HBase region上，产生热点。

Meta Region热点

另一个常见的问题是META region热点。当对HBase集群创建一个连接的时候，应用会先找Zookeeper获取HBase Master和META region的位置。应用将缓存这些信息，然后查询META表，发出读写请求到合适的region中。每次创建一个新的连接，应用都会再次找Zookeeper来获得META。为了避免对ZooKeeper和META表的这些调用，每次执行一个对HBase的请求，推荐创建一个单独的连接，在你的整个应用中共享这个连接。对于Web服务应用，推荐创建一个HBase连接池，在应用侧与所有线程共享这些连接。

防范和解决方案

解决热点问题的最好方法去是防患于未然。在项目之初就要有一个经过妥善测试的行键设计（如果你需要掌握入门知识，可以参考第14章的“行键和表设计”）。同样重要的是，留意你所有的region的度量值，及早发现潜在的热点。在HBase Master的网页接口上，表页面显示了每个表region的请求数。只要它一直在线，Requests列便会显示每个region的读写请求数。当region移至其他的RegionServer上时，或者当一个表被禁用后，region的度量值会被重设。所以当一个region显示出比其他region要高得多的值，就很可能是一个热点，也可能由于两个服务器之间刚刚调整了region平衡。确定是来自热点还是region迁移的唯一方法，是通过浏览日志或者不断地监控可疑region。在生产中避免这个问题最好的方法是对你的应用进行相关测试，让其经历若干开发周期。

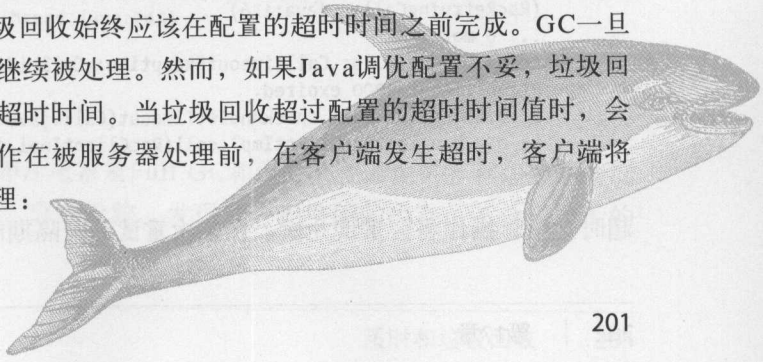
第17章

超时和垃圾回收

垃圾回收是清理Java对象不再使用的内存的进程。这样做是为了释放内存，使之可再次为应用所用。垃圾回收在Java是正常现象，不可避免，但是可以纳入管理之下。根据所使用的垃圾回收算法，可能运行有一个或多个垃圾回收事件。对HBase服务器上影响最大的垃圾回收事件是Java虚拟机需要执行一次彻底的垃圾回收（full GC）。这种操作称做“stop-the-world pause”，当为分配给请求的对象而清理足够的内存的时候，需要暂停JVM。在这个操作期间，当前运行线程将停滞，直到JVM完成了它的GC操作。长时间的GC暂停通常是HBase服务超时的主要原因，但是我们还要研究若干其他的风险。

后果

如上文所述，Full GC是一个“stop-the-world pause”，意味着在HBase上任何运行着的操作将暂停，直至GC完成。这些暂停表现在外部可能是读写延时出现峰值，最糟糕的情况是导致服务器疑似故障。还有一些其他问题的症状与RegionServer的这种性能峰值类似。始终要在处理GC问题前识别产生这些峰值的源头。当执行垃圾回收清理内存时候，JVM将暂停，在此期间所有的读写操作将被客户端归入队列中排队，直到服务端再次响应。垃圾回收始终应该在配置的超时时间之前完成。GC一旦完成，服务器恢复读写操作，继续被处理。然而，如果Java调优配置不妥，垃圾回收的持续时间可能超过配置的超时时间。当垃圾回收超过配置的超时时间值时，会发生一些不同的现象。如果操作在被服务器处理前，在客户端发生超时，客户端将上到异常，则操作将不会被处理：



```
Tue Mar 17 12:23:03 EDT 2015, null, java.net.SocketTimeoutException:\
callTimeout=1, callDuration=2307: row '' on table 'TestTable' at\
region=TestTable,,1426607781080.c1adf2b53088bef7db148b893c2fd4da.,\
hostname=t430s,45896,1426609305849, seqNum=1276
    at o.a.h.a.client.RpcRetryingCallerWithReadReplicas.throwEnrichedException\
    (RpcRetryingCallerWithReadReplicas.java:264)
    at o.a.h.a.client.ScannerCallableWithReplicas.call\
    (ScannerCallableWithReplicas.java:199)
    at o.a.h.a.client.ScannerCallableWithReplicas.call\
    (ScannerCallableWithReplicas.java:56)
    at o.a.h.a.client.RpcRetryingCaller.callWithoutRetries\
    (RpcRetryingCaller.java:200)
    at o.a.h.a.client.ClientScanner.call(ClientScanner.java:287)
    at o.a.h.a.client.ClientScanner.next(ClientScanner.java:367)
    at DeleteGetAndTest.main(DeleteGetAndTest.java:110)
Caused by: java.net.SocketTimeoutException: callTimeout=1, callDuration=2307:\
row '' on table 'TestTable' at region=TestTable,,1426607781080.\
c1adf2b53088bef7db148b893c2fd4da., hostname=t430s,45896,\
1426609305849, seqNum=1276
    at o.a.h.a.client.RpcRetryingCaller.callWithRetries\
    (RpcRetryingCaller.java:159)
    at o.a.h.a.client.ScannerCallableWithReplicas$RetryingRPC.call\
    (ScannerCallableWithReplicas.java:294)
    at o.a.h.a.client.ScannerCallableWithReplicas$RetryingRPC.call\
    (ScannerCallableWithReplicas.java:275)
    at java.util.concurrent.FutureTask.run(FutureTask.java:262)
    at java.util.concurrent.ThreadPoolExecutor.runWorker\
    (ThreadPoolExecutor.java:1145)
    at java.util.concurrent.ThreadPoolExecutor$Worker.run\
    (ThreadPoolExecutor.java:615)
    at java.lang.Thread.run(Thread.java:745)
Caused by: java.io.IOException: Call to t430s/10.32.0.23:45896 failed on local\
exception: o.a.h.a.ipc.CallTimeoutException: Call id=2, waitTime=2001,\
operationTimeout=2000 expired.
    at o.a.h.a.ipc.RpcClientImpl.wrapException(RpcClientImpl.java:1235)
    at o.a.h.a.ipc.RpcClientImpl.call(RpcClientImpl.java:1203)
    at o.a.h.a.ipc.AbstractRpcClient.callBlockingMethod\
    (AbstractRpcClient.java:216)
    at o.a.h.a.ipc.AbstractRpcClient$BlockingRpcChannelImplementation.\
    callBlockingMethod(AbstractRpcClient.java:300)
    at o.a.h.a.protobuf.generated.ClientProtos$ClientService$BlockingStub.scan\
    (ClientProtos.java:31751)
    at o.a.h.a.client.ScannerCallable.call(ScannerCallable.java:199)
    at o.a.h.a.client.ScannerCallable.call(ScannerCallable.java:62)
    at o.a.h.a.client.RpcRetryingCaller.callWithRetries\
    (RpcRetryingCaller.java:126)
    ... 6 more
Caused by: o.a.h.a.ipc.CallTimeoutException: Call id=2, waitTime=2001,\
operationTimeout=2000 expired.
    at o.a.h.a.ipc.Call.checkAndSetTimeout(Call.java:70)
    at o.a.h.a.ipc.RpcClientImpl.call(RpcClientImpl.java:1177)
    ... 12 more
```

超时之前，操作将会重试几次，在每次重试的间隔期间，延迟会加长。

超时同样可能发生在服务端，导致RegionServer疑似故障。HBase RegionServer需要报告给ZooKeeper，确认它们仍然处于活动状态。默认情况下，为了确认他们处于活动状态，当用外部的ZooKeeper的时候，HBase regions服务器需要每隔40秒向ZooKeeper报告一次，或者，当HBase管理ZooKeeper服务时候，需要每隔90秒便报告一次。超时和重试次数可以采用下面的多个参数进行配置：`zookeeper.session.timeout`、`hbase.rpc.timeout`、`hbase.client.retries.number`或`hbase.client.pause`。

当服务器报告太慢，可能导致丢失心跳，给ZooKeeper汇报太晚。当Region服务丢失一个心跳时，会认为被ZooKeeper丢失了，并将汇报给HBase Master，因为无法断定是RegionServer响应太慢，还是服务器已经崩溃了。在发生崩溃的情况下，HBase主机会将先前被分配到服务器上所有的region重新分配到其他的RegionServer上。当重分配region的时候，为了保证一致性，前面等待的操作将重新处理。当响应较慢的RegionServer从暂停中恢复过来，最终向ZooKeeper报告时，RegionServer将被告知其已被认为死亡，并会抛出`YouAreDeadException`异常，最终导致RegionServer终止。

这里是一个服务器长时间停顿导致`YouAreDeadException`异常的一个例子：

```
2015-03-18 12:29:32,664 WARN [t430s,16201,1426696058785-HeapMemoryTunerChore]
      util.Sleeper: We slept 109666ms instead of 60000ms, this is likely due
      to a long garbage collecting pause and it's usually bad, see
      http://hbase.apache.org/book.html#trouble.rs.runtime.zkexpired
2015-03-18 12:29:32,682 FATAL [regionserver/t430s/10.32.0.23:16201] regionserver.
      HRegionServer: ABORTING RegionServer t430s,16201,1426696058785:
      o.a.h.h.YouAreDeadException: Server REPORT rejected; currently
      processing t430s,16201,1426696058785 as dead server
    at o.a.h.h.m.ServerManager.checkIsDead(ServerManager.java:382)
    at o.a.h.h.m.ServerManager.regionServerReport(ServerManager.java:287)
    at o.a.h.h.m.MasterRpcServices.regionServerReport(MasterRpcServices.java:278)
    at o.a.h.h.p.g.RegionServerStatusProtos$RegionServerStatusService$2.
            callBlockingMethod(RegionServerStatusProtos.java:7912)
    at o.a.h.h.i.RpcServer.call(RpcServer.java:2031)
    at o.a.h.h.i.CallRunner.run(CallRunner.java:107)
    at o.a.h.h.i.RpcExecutor.consumerLoop(RpcExecutor.java:130)
    at o.a.h.h.ic.RpcExecutor$1.run(RpcExecutor.java:107)
    at java.lang.Thread.run(Thread.java:745)
```

原因

当在Java平台上处理时，内存碎片通常是Full GC和停顿的主要原因。在理想的情况下，所有的对象是同样的大小，容易追踪、发现和分配空闲的内存，不会造成大的

内存碎片。遗憾的是，实际情况不是这个样子，服务器收到的写入到内存的对象大小几乎从来不会相同大小。同样，压缩的HFiles块和很多其他HBase对象也几乎不会大小相同。先前提到的因素会造成可用内存的碎片化。JVM内存碎片越多，越需要运行GC释放不用的内存。典型的情况下，堆越大，完成Full GC需要的时间越长。这意味着，如果堆太高，不经过合适的优化，可能导致超时。

垃圾回收不是导致超时和停顿的唯一因素。它可能导致内存出席过度分配的场景（分配给HBase或者其他的应用的内存超过了可用物理内存）。操作系统将需要交换一些内存页到硬盘中，这是HBase最不理想的情况。内存交换一定会导致集群问题，进而导致超时，严重影响延迟。

最后的风险是硬件性能失常或故障。网络故障和硬盘设备是此类风险的贴切示例。

存储失败

由于多个原因，存储硬盘可能失败，失去响应。当这种情况发生时，操作系统在报错给应用前将重试多次。这可能导致延误或者全面变缓，进一步影响延时，可能触发超时和故障。HDFS的设计初衷便是处理这类故障，以免丢失数据，但是，操作系统上的故障会影响整个主机的可用性。

节能功能

为降低能耗，一些硬盘有节能功能。当这个功能被触发时，如果硬盘在一段时期内没有被访问，它将进入休眠状态或者停止旋转。一旦系统需要存储在硬盘上的数据，首先需要让硬盘重新开始旋转，等它达到足够的速度，再开始读取这些数据。当HBase对I/O层提出任何请求时，从节能特征的硬盘恢复旋转花费的总共的时间可能导致超时或者延时峰值。在某些情况下，这些超时可能导致服务器失败。

网络错误

在较大的集群上，服务器是布置在机架上的，通过交换机顶层的机架相连。这些交换机非常强大，有很多功能，但像任何其他的硬件一样，它们也会出现一些小的故障，可能导致停顿。这些停顿将导致服务器之间某段时间不能通信。如果这段时间超过的配置的超时时间，HBase将认为这些服务器已经丢失，一旦它们恢复在线，将中止它们，你可以想象对延迟的影响有多大。

解决方案

当某个服务器因为超时问题崩溃或者被关掉的时候，唯一的解决方案是重启。这类故障失败的根源可能是网络原因、配置原因、硬件问题，或者其他的情况。不管故障的根源是什么，都识别并解决，以避免将来发生类似的问题。当你的服务器由于各种原因而出现故障，如果没有解决，将来集群重蹈覆辙的可能性非常高，其他的故障也有可能。识别这些问题最合适的切入点是服务器日志，无论是HBase Master还是有问题的RegionServer。但是，查看系统日志（*/var/log/messages*）也个好办法，如网络接口统计（尤其是丢包）、交换机监控界面等。同时，在不同的机架的多个不同的节点上用iperf等工具验证网络带宽也是不错的做法。测试网络性能和稳定性将使你能检测出配置中的瓶颈，但是可能会漏了配置参数（比如，配置成1Gb/s，而不是10Gb/s等）。

复制硬件和Arista等新一代网络管理工具有助于避免网络和硬件的问题，将在后面章节中探讨。

最后，当RegionServer、Masters和ThriftServers等服务退出的时候，Cloudera Manager和其他的集群管理应用会自动的重启这些服务。然而，如果要了解故障的根本原因，当你在重启服务之前，最好是不要激活这些功能。

有多种方法可以防止这种情况的发生，在下一节中我们逐个地介绍它们。

预防

正如我们刚刚看到的，超时和垃圾回收有多个不同的原因。在这一章中，我们要确定这些情况可能是什么，并提供一些防范方法。

减小堆大小

堆越大，JVM完成GC的时间越长。即使有更多的内存可以用来增加缓存大小，改善延迟和性能，但是因为GC，你可能遇长更长的停顿。由于JVM默认的垃圾回收算法，彻底的垃圾回收操作将冻结应用程序，很可能导致ZooKeeper超时，进而RegionServer被中止。在HBase Master服务器上，GC暂停的风险很小。在默认的HBase设置下，HBase堆大小推荐在20GB以下。垃圾回收可以在*conf/hbase-conf.sh*脚本中配置。默认设置是采用Java CMS（Concurrent MarkSweep）算法：`export HBASE_OPTS="-XX:+UseConcMarkSweepGC"`。

在配置文件中有关于GC过程的其他调试信息。下面的示例中，我们只增加其中的一个选项（我们推荐查看一下配置文件，了解所有这些可用的选项）：

```
# This enables basic GC logging to its own file with automatic log rolling.
# Only applies to jdk 1.6.0_34+ and 1.7.0_2+.
# If FILE-PATH is not replaced, the log file(.gc) would still be generated in
# the HBASE_LOG_DIR .
# export CLIENT_GC_OPTS="-verbose:gc -XX:+PrintGCDetails -XX:+PrintGCDateStamps\
# -Xloggc:<FILE-PATH> -XX:+UseGCLogFileRotation -XX:NumberOfGCLogFiles=1\
# -XX:GCLogFileSize=512M"
```

从Oracle JDK 1.7开始，JVM提供了一个新的垃圾回收算法，称为G1（Garbage First）。本章后面的“使用G1GC算法”中有关于G1的详细介绍。

堆外读缓存

HBase 1.0或以上的版本允许你使用BucketCache配置堆外读缓存。使用堆外读缓存，能使HBase自己管理碎片。由于所有的HBase块被认为是同样的大小（除非在表级别特意另行定义），所以管理相关内存碎片并非难事。这样能减少堆内存的使用和内存碎片，进而使得垃圾内存变少。当用堆外缓存的时候，你可以减小默认的堆大小内存，这样能减少JVM运行Full GC所需的时间。同样，如果HBase正在承受的负载有读也有写，由于只有读缓存被移出堆外，你可以减小堆内读缓存，使用`hbase.regionserver.global.memstore`和`hfile.block.cache.size`属性分给memstore更多的内存。当服务器内存有限时，可以配置BucketCache，将SSD硬盘等特定设备作为缓存的内存存储。

使用G1的GC算法

Java6、7和8中默认的JVM GC算法倾向于大堆运行Full GC。这些Full GC将暂停JVM直到内存清空完毕。正如先前的章节看到的，这可能对服务器的可用性有负面影响。Java 1.7_u60+是G1 GC算法第一个稳定的发布版本。G1 GC是一种垃圾回收算法，代表“垃圾优先（Garbage-first）”的意思。根据堆的大小，G1将内存划分成1到32MB大小的区域。然后当运行cleanup，试图先清理不再使用的区域，G1便释放了更多的内存，而不必执行stop-the-world暂停操作。G1 GC在快速的写入过程中也大量使用混合方式收集内存，以避免进入对HBase延迟有害的stop-the-world暂停操作。

调优G1 GC收集器并非易事，需要不断尝试。一旦配置妥当，G1 GC收集器的堆可以大到176GB。较大的堆不只是提高读取性能那么简单。如果你就是要提高读性

能，堆外读缓存也可以提供好的性能，还不用面临调优的烦恼。G1 GC堆较大的话，能给regions提供一个较大的memstore池共享内存，HBase水平扩展性更好。

必用参数

- `-XX:+UseG1GC`
- `-XX:+PrintFlagsFinal`
- `-XX:+PrintGCDetails`
- `-XX:+PrintGCDateStamps`
- `-XX:+PrintGCTimeStamps`
- `-XX:+PrintAdaptiveSizePolicy`
- `-XX:+PrintReferenceGC`

当堆超出100GB时，HBase的额外设置

`-XX:-ResizePLAB`

当每个GC期间， PLABs (Promotion local allocation buffers)会避免GC线程之间出现大的通信开销以及变化。

`-XX:+ParallelRefProcEnabled`

当这个参数立起时，在做新生代GC和混合GC期间，GC会利用多个线程处理越来越多的引用。

`-XX:+AlwaysPreTouch`

在JVM初始化时，把Java堆预先摸底。这意味着堆的每一页在初始化的时候就将要求清零，而不是在应用运行的时候递增。

`-XX:MaxGCPauseMillis=100`

最长暂停会尝试将GC暂停保持在参考时间以内。这对于正确调优变得至关重要。 G1 GC将尽力控制在限值以下，但如有必要，会暂停更长时间。

其他有意思的参数

`-XX:ParallelGCThreads=X`

计算公式：8 + (逻辑处理器个数 - 8) (5/8) 。

`-XX:G1NewSizePercent=X`

Eden代的大小 = 堆大小 × G1NewSizePercent。G1NewSizePercent默认的值是5（堆总值的5%）。降低这个值会改变完成新生代GC暂停所需要的总时间。

`-XX:+UnlockExperimentalVMOptions`

解锁前面引用的旗标。

`-XX:G1HeapWastePercent=5`

设置你容许浪费的堆百分比。如果可回收百分比小于堆废物百分比，Java HotSpot VM 不会启动混合垃圾回收周期。默认值是 10%。

`-XX:G1MixedGCLiveThresholdPercent=75`

为混合垃圾回收周期中包括的旧区设置占用率阈值。默认占用率为 65%。这是一个实验性的旗标。

调优的首要目标是运行多个混合垃圾回收，避免Full GC。一旦调优合适，只要负载保持相对的统一，GC操作应该是可重复的。值得注意的是，堆的过度配置，将余量留给混合GC让其在后台操作，避免Full GC。欲详细阅读，查看Oracle的综合文档。

配置Swappiness为0或者1

HBase可能不是在你系统上运行的仅有应用，在某一时刻，内存可能被过度分配。在默认情况下，Linux会对缺少内存未雨绸缪，开始将内存页交换给硬盘。这样做是没有问题的，因为以让操作系统避免内存不足，但是，将这些内存页写至硬盘会花费一定的时间，同样将影响延迟。当有大量内存需要通过这种方法写入硬盘中，操作系统可能会发生明显的停顿，将导致HBase服务器失去ZooKeeper的心跳而中止。为避免这个情况，建议降低swappiness至最小值。根据Linux内核的版本的不同，这个值会被设置成0或者1。自3.5-rc1以上的内核版本起，`vm.swappiness`设为1；早期版本的内核，`vm.swappiness`设为0。此外，不要允许过载使用内存。对于每一个运行在你服务器上的应用，要留意被分配的内存情况（如果有的话，也包括堆外读缓存），求其和。当你清单部分为操作系统保留的部分内存后，你需要将这个值保持在可用的物理内存以下。

128GB服务器内存分配示例

- 操作系统保留2GB。
- Kerberos客户端、SSSD等本地的应用保留2GB。
- DataNode节点保留2GB。
- 操作系统缓存保留2GB。
- 剩下的120GB可分配给HBase。

但是，当分配的堆内存超过20GB时，GC可能会是一个问题，你可以配置HBase，使用20GB的堆内存加100GB的堆外读缓存。

禁用环境友好功能

正如本章中我们前面所讨论的，环保节能等特性会影响服务器的性能和稳定性。建议关闭所有的这些特性。通常在BIOS中可以关掉它们。但是，有些硬盘有时候不能简单地关掉这些特性。如果碰到这种情况，建议更换这些硬盘。在生产环境中，你需要谨慎订购这些硬盘，避开包含这些特性的任何东西。

硬件复制

为避免停机而复制硬件是常见的好方法。主要的例子有：在RAID中运行两个OS 驱动盘；利用双电源供电；在配置故障转移中绑定网络接口。这些复制操作将清除无数的单个故障点，使集群正常运行时间更长，维护操作更少。这同样能应用于网络交换机和集群中的所有其他的硬件。

第18章

HBCK和不一致

HBase文件系统布局

和任何数据库和文件系统一样，HBase在其理想的元数据情况和它实际的文件系统情况之间会陷入不一致。当然，前面的说法反过来也可能成立。在开始调试HBase不一致之前，重要的是要理解被称为hbase:meta的HBase元数据主表结构，以及HBase是如何布置在HDFS上的。我们来看称为hbase:meta的元数据表，“：”之前的hbase表示表所在的名字空间，“：”之后的是表名，即meta。名字空间用于对相似的表进行逻辑分组，一般在多租户环境中使用。有两个现成的名字空间可以使用：default和hbase.default，所有没有指定名字空间的表即为default，而hbase用于HBase的内部表。现在，我们主要研究hbase:meta表。HBase的元数据表用来在HBase表中存储region的重要分片信息。下面是名为odell用户表的HBase实例输出的示范：

```
hbase(main):002:0> describe 'hbase:meta'
DESCRIPTION
  'hbase:meta', {TABLE_ATTRIBUTES => {IS_META => 'true', coprocessor$1 =>
  '|org.apache.hadoop.hbase.coprocessor.MultiRowMutation Endpoint|536870911|'},
  {NAME => 'info', DATA_BLOCK_ENCODING => 'NONE', BLOOMFILTER => 'NONE',
  REPLICATION_SCOPE => '0', COMPRESSION => 'NONE', VERSIONS => '10', TTL =>
  'FOREVER', MIN_VERSIONS => '0', KEEP_DELETED_CELLS => 'false', BLOCKSIZE =>
  '8192', IN_MEMORY => 'true', BLOCKCACHE => 'true'}
```

其中收集到一些如下重要信息：

IS_META =>true
: 这是指你描述的表是元数据表，希望这不会令人惊讶。

NAME =>*info*

在meta表中只有一个叫做info的列族。我们下面会进一步查看存储在这个列族下面的其他信息。

IN_MEMORY =>*true*, *BLOCKCACHE* =>*true*

元数据表和HBase的索引都存储在块缓存中，记住从不要将块缓存设置在必要的值（通常，堆的10%足矣）以下。

查看META表

看下面的数据块，元数据表本身读起来不是多么有趣，但是HBase不一致问题跟它离不开干系：

```
hbase(main):010:0> scan 'hbase:meta', {STARTROW => 'odell,,'}
ROW                                                    COLUMN+CELL

odell,,1412793323534.aa18c6b576bd8fe3eaf71382475bade8.
column=info:regioninfo, timestamp=1412793324037, value={ENCODED => aa18c6b576b...
column=info:seqnumDuringOpen, timestamp=1412793324138, value=\x00\x00\x00\x00\...
column=info:server, timestamp=1412793324138, value=odell-test-5.ent.cloudera.c...
column=info:serverstartcode, timestamp=1412793324138, value=1410381620515

odell,ccc,1412793397646.3eadbb7dcbfeee47e8751b356853b17e.
column=info:regioninfo, timestamp=1412793398180, value={ENCODED => 3eadbb7dcbf...
column=info:seqnumDuringOpen, timestamp=1412793398398, value=\x00\x00\x00\x00\...
column=info:server, timestamp=1412793398398, value=odell-test-3.ent.cloudera.c...
column=info:serverstartcode, timestamp=1412793398398, value=1410381620376

...truncated
```

输出的第一列是行键：

```
odell,,1412793323534.aa18c6b576bd8fe3eaf71382475bade8.
```

然后:

```
odell,ccc,1412793397646.3eadbb7dcbfeee47e8751b356853b17e.
```

行键被分解成表名、起始行键、时间戳、编码的region名和“.”（没错，“.”必不可少）。当故障排除时，最重要的方面是表名、编码的region名和起始行键，它们理应匹配。下面的列是这个行的键值对。在本例中，有一个称为info的列族，四个分别称为regioninfo、seqnumDuringOpen、server和serverstartcode的列。

当查看meta表特定region的时候，有几个主要的值需要注意：

`info:regioninfo`

包含了编码的region名、行键、行键的起始值和行键的结束值。

`info:seqnumDuringOpen`

用于后面的shadow regions等HBase的特性，但对当前故障排除不重要。

`info:server`

包含了 RegionServer的信息， region被分配至该RegionServer（对未分配的regions进行故障排除时，这非常有用）。

`info:serverstartcode`

包含了Region服务上特定region的开始时间。

在HDFS上查看HBase

查看meta表的结构是了解HBase region结构应该如何的好方法。但是有时候，meta表可能误导或者不正确。在这种情况下，HDFS才是HBase信息的正确的来源。查看meta表和查看HBase的HDFS文件结构同等重要。我们现在开始探索硬盘上HFFS的布局：

```
-bash-4.1$ hadoop fs -ls /hbase
Found 9 items
drwxr-xr-x   - hbase hbase          0 2014-10-08 12:14 /hbase/.hbase-snapshot
drwxr-xr-x   - hbase hbase          0 2014-08-26 10:36 /hbase/.migration
drwxr-xr-x   - hbase hbase          0 2014-09-30 06:48 /hbase/.tmp
drwxr-xr-x   - hbase hbase          0 2014-09-10 13:40 /hbase/WALs
drwxr-xr-x   - hbase hbase          0 2014-10-10 21:33 /hbase/archive
drwxr-xr-x   - hbase hbase          0 2014-08-28 08:49 /hbase/data
-rw-r--r--   3 hbase hbase         42 2014-08-28 08:53 /hbase/hbase.id
-rw-r--r--   3 hbase hbase          7 2014-08-28 08:49 /hbase/hbase.version
drwxr-xr-x   - hbase hbase          0 2014-10-14 06:45 /hbase/oldWALs
```

以.开头的第一组目录都是HBase内部目录，不包含任何数据。*.hbase-snapshot*目录顾名思义，它包含了HBase所有当前快照信息；接下来是*.migration*目录，在更新HBase版本到下一个版本的时候有用；*.tmp*目录被用来存放创建的临时文件。此外，FSUtils和HBCK将利用这个空间，比如，*hbase.version*文件就在这个目录里创建，然后一旦写完，被移动到*/hbase*目录，当合并region，或任何涉及到文件系统布局变动的操作的时候，*HBCK*都将用到这个目录；WAL目录包含所有当前活动的WAL，以及还没有启动或者在重启时候需要分裂的WAL；*archive*目录直接和*.hbase-snapshots*相关，只用于快照功能。*archive*目录直接包含为HBase快照所保护（而非为其所删除）的HFile文件；*hbase.id*文件包含了集群的唯一ID；*hbase.version*文件

保存了表示当前HBase版本的字符串；*oldWALs*目录和HBase复制操作直接关联，任何需要在目标集群上重新起作用的WALs都会被写到这里，而不是在恢复的时候删除它们。在源集群和目标集群之间发生通信问题时，通常出现这种备份。为了定位不一致问题，我们着重于*data*目录，*data*目录命名恰当，它包含了HBase的数据。让我们深入了解下*data*的目录结构：

```
-bash-4.1$ hadoop fs -ls /hbase/data
Found 2 items
drwxr-xr-x     - hbase hbase          0 2014-10-08 11:31 /hbase/data/default
drwxr-xr-x     - hbase hbase          0 2014-08-28 08:53 /hbase/data/hbase
```

下一层包含了HBase名字空间。在先前的示例中，仅有两个名字空间default和hbase:

```
-bash-4.1$ hadoop fs -ls /hbase/data/default
Found 1 items
drwxr-xr-x    - hbase hbase          0 2014-10-08 11:40 /hbase/data/default/odell
-bash-4.1$ hadoop fs -ls /hbase/data/hbase
Found 2 items
drwxr-xr-x    - hbase hbase          0 2014-08-28 08:53 /hbase/data/hbase/meta
drwxr-xr-x    - hbase hbase          0 2014-08-28 08:53 /hbase/.../namespace
```

接下来的一层显示了表名。在先前的代码片段中，默认的名字空间中有一个称为“odell”的表，在hbase名字空间中，有`meta`和`namespace`两个表。接下来我们着重了解“odell”表看起来是什么样子：

```
-bash-4.1$ hadoop fs -ls /hbase/data/default/odell
Found 5 items
drwxr-xr-x 2014-10-08 11:31 /hbase/data/default/odell/.tabledesc
drwxr-xr-x 2014-10-08 11:31 /hbase/data/default/odell/.tmp
drwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/3eadbb7dcbfeee47e875...
drwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/7450bb77ac287b9e77ad...
drwxr-xr-x 2014-10-08 11:35 /hbase/data/default/odell/aa18c6b576bd8fe3eaf71...
```

*.tabledesc*目录包含了一个通常称为“*.tableinfo.000000xxxx*”的文件，其中*x*指表的数量。*tableinfo*文件包含了在HBase shell上运行`describe`时看到的相同的信息，包含了meta、数据块的编码类型、使用中的布隆过滤器、版本数、压缩方式等。当试图用distcp（我们更推荐用snapshots）复制表的时候，重要的是要维护好tableinfo表。

*.tmp*目录用于写入*tableinfo*文件，当写完的时候，会移至上面的*.tabledesc*目录中。接下来，是经过编码的region名字。回顾下meta表的输出情况，可以发现，经过编码的region名字应该和之前meta表中的`info:regioninfo`里经过编码的region名字相匹配。在每个经过编码的region目录之下，是这样的：

```
-bash-4.1$ hadoop fs -ls -R /hbase/data/default/odell/3ead...
-rwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/3ead.../.regioninfo
drwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/3ead.../.tmp
drwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/3ead.../cf1
-rwxr-xr-x 2014-10-08 11:36 /hbase/data/default/odell/3ead.../cf1/5cadc83fc35d...
```

*.regioninfo*文件包含了关于region的<INFORMATION>信息；在单个region层面上，*.tmp*目录在大型合并期间用于重写storefiles；最后，表中每个列族会有一个目录，如果有任何数据被写到该列族的硬盘上，都会包含storefiles文件。

HBCK概述

现在我们对meta和文件系统上的HBase内部有了理解，让我们看看当一切完好无损的时候，从逻辑角度而言HBase看上去是什么样子。在先前的示例中有一个表称为“odell”，它有“–aaa,aaa–ccc, ccc–eee, eee–”三个region。将这些数据可视化会很有帮助：

表18-1：HBCK 数据可视化

Region1	Region2	Region3	…	Region24	Region25	Region26
"	aaa	bbb	…	xxx	yyy	zzz
aaa	bbb	ccc	…	yyy	zzz	"

假设有一个包含字母的HBase表，表18-1是它的逻辑图。每一组行键被分配到各自的region中，以引号代表开始和结束，捕获行键前后所有内容。

早期版本的HBase，由于分裂错误、合并失败和region清除不彻底，容易出现不一致的情况。后面的HBase版本相当稳定，很少遇到不一致。但是软件和生活一样，这个世界上没有什么是一定的，是软件就会有错误。最好是时候有准备。修复HBase不一致的首选工具称为HBCK工具。这个工具能够修复你遇到的大部分HBase问题。HBCK工具可以通过在控制台上运行hbase hbck来执行：

```
-bash-4.1$ sudo -u hbase hbase hbck
14/10/15 05:23:24 INFO Client environment:zookeeper.version=3.4.5-cdh5.1.2--1,...
14/10/15 05:23:24 INFO Client environment:host.name=odell-test-1.ent.cloudera.com
14/10/15 05:23:24 INFO Client environment:java.version=1.7.0_55
14/10/15 05:23:24 INFO Client environment:java.vendor=Oracle Corporation
14/10/15 05:23:24 INFO Client environment:java.home=/usr/java/jdk1.7.0_55-clou...
...truncated...
Summary:
hbase:meta is okay.
Number of regions: 1
Deployed on: odell-test-5.ent.cloudera.com,60020,1410381620515
odell is okay.
```

```
Number of regions: 3
Deployed on: odell-test-3.ent.cloudera.com,60020,1410381620376
hbase:namespace is okay.
Number of regions: 1
Deployed on: odell-test-4.ent.cloudera.com,60020,1410381620086
0 inconsistencies detected.
Status: OK
```

前面的代码呈现了一个理想的HBase实例，所有region都有被分配，META正确无误；HDFS上所有的region信息也是正确的；所有的regions当前是一致的。如果一切都顺利运行，应该不会出现不一致的情况，万事安好。有些情况会导致HBase崩溃。我们来深入看一下这些比较常见的场景：

- region分配不当。
- META损坏。
- HDFS空洞。
- HDFS上region被孤立。
- Region重叠。

使用HBCK

当处理不一致问题，通常会出现错误的反馈，导致情形看上去比真正情况要更为糟糕。例如，META文件损坏，可能引起很多HDFS重叠或者出现空洞，然后事实上底层文件系统安然无恙。在HBCK中运行的最重要的参数是-repair。这个参数将在一行命名中执行所有的修复操作：

```
-fixAssignments
-fixMeta
-fixHdfsHoles
-fixHdfsOrphans
-fixHdfsOverlaps
-fixVersionFile
-sidelineBigOverlaps
-fixReferenceFiles
-fixTableLocks
```

当你处理的是实验实例或者开发实例的时候，这固然不错，但是处理生产实例或者预生产实例时，就不是那么理想。谨慎执行`-repair`参数的一个主要原因是`-sidelineBigOverlaps`参数。如果重叠过大，HBase会将region下线，它们不得不批量加载，重新进行region分配。如果不充分理解每一个参数的含义，很可能让问题变得更糟糕。建议采用务实的方法，从影响较小的参数入手。

用日志记录一切

在开始运行HBCK之前，确保你已是在记录至外部文件，或者你的终端在记录所有命令和终端输出。

通常，我们最喜欢运行的两个参数是-fixAssignments和-FixMeta。-fixAssignments参数会修复所有未分配的、分配不当的或者重复分配的region。HBase利用HDFS作为META正确布局的根本来源。当HDFS中没有相应的region时，-fixMeta参数会删除meta行；当HDFS中有region，但是不在META中时，-fixMeta参数会增加新的meta行。在HBase中，region分配由分配管理器控制。分配管理器在内存中存储HBase当前的状态，如果region分配与HBase和META不同步，可以肯定地认为它们在分配管理器中也不同步。通过HBCK回滚重启HBase主节点，是将分配管理器更新到正确值的最快方法。重启HBase主节点后，就可以再次运行HBCK。

如果重新运行HBCK以后，最终结果不是“0 inconsistencies detected”，那么该使用一些强大的命令矫正这些悬而未决的问题。有三个重要的问题可能还会发生，分别是HDFS空洞、HDFS重叠和HDFS孤立。

如果运行-FixMeta和-FixAssignments参数，我们建议联系附近的Hadoop提供商了解详细的使用说明。另外，如果你自己动手处理，我们建议在这个时候使用-repair参数。重要的是，要注意可以通过许多途径。我们建议像采用下面的循环重复运行-repair参数：

```
-bash-4.1$ sudo -u hbase hbase hbck

-bash-4.1$ sudo -u hbase hbase hbck -repair

-bash-4.1$ sudo -u hbase hbase hbck

-bash-4.1$ sudo -u hbase hbase hbck -repair

-bash-4.1$ sudo -u hbase hbase hbck

-bash-4.1$ sudo -u hbase hbase hbck -repair

-bash-4.1$ sudo -u hbase hbase hbck
```

如果你运行完这组命令，仍然存在不一致的情况，你可能需要开始运行若干单个命令，具体取决于最后一条HBCK命令的输出结果。同样，在这个时候，我们还是强调要联系你的Hadoop提供商或者查看Apache邮件列表，那里有专家可以帮助解决这种情形的问题。以下一份列表，举例了HBCK中的其他命令：

`-fixHdfsHoles`

尝试修复HDFS中的空洞。

`-fixHdfsOrphans`

尝试修复HDFS中没有*.regioninfo*文件的region目录。

`-fixTableOrphans`

尝试修复HDFS中没有*.tableinfo*文件的表目录（仅在线模式下）。

`-fixHdfsOverlaps`

尝试修复HDFS中region重叠的问题。

`-fixVersionFile`

尝试修复HDFS中缺少*hbase.version*文件的问题。

`-sidelineBigOverlaps`

修复region重叠问题时，允许将严重重叠的部分下线。

`-fixReferenceFiles`

尝试将延迟的引用存储文件下线。

`-fixEmptyMetaCells`

尝试修复`hbase:meta`不引用任何的region（`REGIONINFO_QUALIFIER`空行）的接口。

`-maxMerge <n>`

当修复region重叠问题时，最多允许`<n>`个region合并(n默认值为5)。

`-maxOverlapsToSideline <n>`

当修复region重叠问题时，每组最多允许`<n>`个region下线（n默认值为2）。

上面所列的并不全面，当然也不打算巨细无遗。Meta文件和底层的HDFS结构出岔子时，有很多拦路虎在前面，还需你独自解决。

作者介绍

Jean-Marc Spaggiari，2012开始一直是HBase源码的贡献者，在Cloudera公司作为HBase的专业解决方案架构师，负责Hadoop和HBase技术支持和咨询相关工作，服务着北美最大的HBase用户群。

Jean-Marc主要支持HBase用户部署、升级、配置和优化集群，同时支持它们在HBase相关的应用开发。他也是一个很活跃的HBase社区成员，从性能和稳定性角度测试每一个HBase发布的版本。

在加入Cloudera之前，Jean-Marc担任项目经理，同时也是CGI和保险公司解决方案的架构师。他有着将近20年的Java开发经验。除了定期的参与HBaseCon，也在各种北美的Hadoop用户组的meetup和会议上做分享，内容主要聚焦在HBase的相关介绍和展示。

Kevin O'Dell，自2012开始一直是HBase的贡献者，并一直活跃于社区。Kevin在各种大数据会议和活动上发表过演讲，包括Hadoop User Groups, Hadoop Summits, Hadoop Worlds和HBaseCons等。目前在Rocana公司做销售工程师，在这个工作中，当生产环境中发生各种高级的异常问题时，Kevin主要去解决大规模监控问题。Kevin之前在Cloudera担任系统工程师，专门构建HBase相关的大数据应用，在这个工作中，Kevin设计、估算和部署了各种垂直行业的大数据应用。Kevin 也曾在Cloudera的技术支持团队工作过，在那里，他作为团队负责人支持了当时已知的世界最大规模的HBase部署 。

在加入Cloudera之前，Kevin在EMC 的数据部门工作，作为Hardware/RAID/SCSI团队的全球支持负责人，他管理着一个国际化的团队支持了许多世界财富500强的企业客户。Kevin还曾任职于NetApp，借助WAFL文件系统，专注于NetApp在SAN和NAS部署上的性能问题。

封面介绍

本书的封面是一只杀人鲸或逆戟鲸（虎鲸）。虎鲸有黑色和白色，包括眼睛上方特有的白色斑块。公鲸身长可达26英尺，体重可达6吨。母鲸略小，能长到23英尺和4吨大小。

虎鲸是有牙齿的鲸鱼，并以鱼类、海洋哺乳动物、鸟类甚至鲸鱼为食。在它们的生态系统中，它们是顶尖捕食者，也就是说它们没有天敌。虎鲸群（称为pods）已被观察到专门吃什么，所以每个虎鲸的饮食各有不同。虎鲸是高度群居的动物，并发展出了复杂的关系和层次结构。众所周知，它们一代代地传授知识，如狩猎技巧和发声。随着时间的推移，这将产生不同的虎鲸群之间的发散行为的效果。

虎鲸并没有被列为对人类的威胁，长期以来在一些文化神话中起到了一定的作用。像大多数种类的鲸鱼一样，由于商业捕捞，虎鲸的数量在过去的几个世纪里大大减少了。虽然捕鲸已被禁止，但虎鲸仍然受到人类活动的威胁，包括船只碰撞和捕鱼线纠缠。目前，虎鲸的数量是未知的，但估计是50000左右。

O'Reilly封面的许多动物都是濒临灭绝的，它们对于这个世界上都是很重要的。想要了解更多的关于你如何帮助它们，访问*animals.oreilly.com*。

封面图片来自于英国Quadrapeds。

O'REILLY®

HBase应用架构

当对大量数据构建索引的时候，HBase是一个出色的工具，但是从零开始学习分布式数据库及其生态系统是一件让人望而却步的事情。通过手把手的指导和对真实环境中案例的阐释，你将学会如何架构、设计，以及部署你自己的HBase应用程序。除了HBase原理和集群部署指南之外，本书通过对案例的深入研究，展示了大型企业如何利用HBase解决具体问题的用例。

本书提供了基本的解决方案和代码示例来帮助你实现自己的用例，包括主数据管理（MDM）和文件系统，以及准实时事件处理。你也能学习使用故障排除的方法来帮你避免部署时出现的一些问题。

- 学习HBase能用来做什么，其生态系统包括哪些组件以及如何搭建你的环境。
- 探索现实世界中HBase实例如何部署并投入生产环境。
- 查验用于追踪监控索赔的记录用例，并诊断数据管理以及产品质量。
- 理解HBase如何和Spark、kafka、MapReduce，以及Java API一起使用。
- 学习如何识别最常见的HBase问题，并理解其结果。

“本书由HBase部署的专业团队编写。Jean-Marc和Kevin了解这其中的一切知识。读这本书并向最出色的工程师学习。”

——Michael Stack
ApacheHBase PMC

Jean-Marc Spaggiari，自2012年来是HBase的contributor，作为Cloudera 的HBase精通解决方案架构师，他一直从事着Hadoop和HBase的技术支持和咨询工作。他曾经与北美洲一些最大的HBase用户一起工作。

Kevin O'Dell，自2012年来是HBase的contributor，作为Rocana的现场工程师，他和客户一起设计并完成大规模的IT运营。此外，他还在HBaseCon、HadoopSummit及一些Hadoop用户组做过分享。

DATA | HADOOP | HBASE

O'Reilly Media, Inc.授权中国电力出版社出版

ISBN 978-7-5198-1121-1
9 787519 811211 >
定价：48.00元